The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.

– Albert Einstein

I hear and I forget. I see and I remember. I do and I understand.

– Confucius

**University of Alberta**

# SoC INTERCONNECTION BUS DESIGNS USING MIXED-CLOCK FIFO

by

**Edmund Fung** ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2005

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-494-09167-3*
*Our file Notre référence*
*ISBN: 0-494-09167-3*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

I would like to dedicate this work to my parents, brother, sister and Lian who have supported and encouraged me throughout my studies

# Abstract

In this thesis, we propose two novel mixed-clock on-chip communication networks for *system-on-a-chip* (SoC) that operate using a token passing scheme. An on-chip interconnection network that provides interfaces among mixed-timing modules is one of the most important part of future SoC designs. The token passing mechanism of our mixed-clock interconnection networks is embedded in an element called a Mixed-Clock FIFO. Mixed-clock shared bus structures are then constructed using these Mixed-Clock FIFOs. The shared buses provide high performance multi-point interconnection and data broadcasting among mixed-clock modules in an SoC, and can be constructed using standard library components. One of the shared bus designs, synthesized using 0.18$u$m CMOS technology at 1.6V and 300 K, provides a throughput of 416MB/s with a maximum latency of 8.33ns. That is comparable to the performance of other synchronous on-chip buses.

# Acknowledgements

This research was funded by research grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), and equipment loans from the Canadian Microelectronics Corporation and the University of Alberta. I would like to thank my supervisor Dr. Stephen Bates for providing opportunities for in-depth research on this thesis topic and for his guidance and suggestions throughout my thesis research. I like to thank Dr. Xiaoling Sun for acquiring funding for my research. I would also like to thank Dr. Tiberiu Chelcea from the Carnegie Mellon University and Dr. Steven M. Nowick from the Columbia University for answering my questions on their publications. In addition, I wish to thank Dave Nguyen, Yoko Aoki, Tyler Brandon, Amir Alimohammad, Madhura Purnaprajna, Nitin Parimi, Kaston Leung, Christian Giasson, John Koob, and Kris Breen for their support and assistance throughout my thesis research. Without the efforts of these people, the completion of this thesis would have been immensely difficult. Finally, I must thank Lian Wen Huang for her patience, understanding and encouragement.

# Table of Contents

**1 Introduction** **1**

1.1 Thesis Organization . . . 3

**2 Mixed-clock Interface for SoC** **5**

2.1 Synchronous Design's Challenges . . . 5
  2.1.1 Global Clock . . . 6
  2.1.2 Clock Generation and Distribution . . . 7
  2.1.3 Synchronous Design Limitations . . . 8
2.2 Asynchronous Design Scheme . . . 9
  2.2.1 Purely Asynchronous circuits . . . 10
  2.2.2 Handshaking Protocols . . . 10
    2.2.2.1 Micropipelines . . . 12
  2.2.3 Data Encoding . . . 13
    2.2.3.1 Single-rail Encoding . . . 13
    2.2.3.2 Dual-rail Encoding . . . 15
    2.2.3.3 Dual-rail Pipeline . . . 16
    2.2.3.4 1-of-n and m-of-n Encoding . . . 18
    2.2.3.5 Color Coding . . . 19
  2.2.4 Self-timed Systems . . . 20
  2.2.5 Disadvantage of Asynchronous Systems . . . 21
2.3 Mixed-clock System . . . 22
  2.3.1 Mixed-clock Interface . . . 25
    2.3.1.1 Globally-Asynchronous Locally-Synchronous (GALS) Systems . . . 26
    2.3.1.2 Synchronization by Frequency Matching . . . 28
    2.3.1.3 Data Synchronization System . . . 31
  2.3.2 Synchronizer and Mixed-Clock FIFO . . . 32
    2.3.2.1 Synchronization Issues . . . 33
    2.3.2.2 Latency . . . 36
2.4 Summary . . . 38

**3 Mixed-Clock FIFO** **41**

3.1 Mixed-clock Interface . . . 41
3.2 Design of the Mixed-Clock FIFO (MCFIFO) . . . 44

3.2.1 FIFO Architecture . . . . . . . . . . . . . . . . . . . . . . 46
3.2.2 FIFO Cell's Implementation . . . . . . . . . . . . . . . . . 53
3.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**4 Shared Bus Design Using MCFIFO** **55**
4.1 Broadcasting of Data Items Using the MCFIFO . . . . . . . . . . . 55
4.1.1 Shift Register . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.1.2 DBFIFO Protocol . . . . . . . . . . . . . . . . . . . . . . . 60
4.1.3 DBFIFO Applications . . . . . . . . . . . . . . . . . . . . . 63
4.1.4 Shared Bus . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.2 Data Broadcasting Shared Bus (DBSB) . . . . . . . . . . . . . . . 65
4.2.1 Bus Controller . . . . . . . . . . . . . . . . . . . . . . . . 67
4.2.2 Shift Register . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.2.3 Bus Traffic Control . . . . . . . . . . . . . . . . . . . . . . 68
4.2.4 DBSB Protocol . . . . . . . . . . . . . . . . . . . . . . . . 69
4.3 Mixed-Clock Shared Bus (MCSB) . . . . . . . . . . . . . . . . . 70
4.3.1 MCSB Protocol . . . . . . . . . . . . . . . . . . . . . . . . 70
4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

**5 Design and Performance Evaluation** **73**
5.1 Test Models of Mixed-clock Buses . . . . . . . . . . . . . . . . . 73
5.2 Performance Analysis . . . . . . . . . . . . . . . . . . . . . . . 75
5.3 Comparison with Synchronous Buses . . . . . . . . . . . . . . . . 83
5.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**6 Conclusion** **87**
6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

**A Verilog Code** **95**
A.1 Synthesizable Verilog Model of DBSB (using a 4-place MCFIFO) . 95
A.2 Synthesizable Verilog Model of MCSB (using a 4-place MCFIFO) . 118

**B Simulation Waveforms** **131**
B.1 Simulations of DBSB . . . . . . . . . . . . . . . . . . . . . . . . 131
B.2 Simulations of MCSB . . . . . . . . . . . . . . . . . . . . . . . . 131

# List of Tables

2.1 Muller C-element's Truth Table . . . . . . . . . . . . . . . . . . . . . . 16

3.1 Get Operation Outcomes . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1 Timing Relationships between The Three Subsystems . . . . . . . . . 58

5.1 DBSB Functional Test Cases . . . . . . . . . . . . . . . . . . . . . . 75
5.2 Performance Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.3 Performance comparisons of Synchronous Buses [3,4] . . . . . . . . 84

# List of Figures

2.1 Two-phase Signaling . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Four-phase Signaling . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Micropipelines . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Dual-rail Circuit Implemented using DIMS . . . . . . . . . . . . . 17
2.5 Asynchronous Latch . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.6 Dual-rail Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.7 Various Types of Multiple Clock Domains . . . . . . . . . . . . . . 23
2.8 Mesochronous Clocking . . . . . . . . . . . . . . . . . . . . . . . 24
2.9 Pausible Clock Generator . . . . . . . . . . . . . . . . . . . . . . 27
2.10 Asynchronous Wrapper and Configuration of a Data Channel in GALS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.11 Single-Stage Frequency Matching FIFO . . . . . . . . . . . . . . . 28
2.12 Timing Diagram of Single-Stage Frequency Matching FIFO . . . . 29
2.13 Latch Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.14 Implementation of a Frequency Matching FIFO Interface . . . . . . 31
2.15 Two Flip-flip Synchronizer . . . . . . . . . . . . . . . . . . . . . . 33
2.16 Metastability in Data Transaction . . . . . . . . . . . . . . . . . . 34
2.17 Implementation of a Two Flip-flop Synchronizer Data Channel . . . 37
2.18 FSM Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.1 Mixed-clock Interconnection with Two Data Registers . . . . . . . 42
3.2 The Full and The Empty Generators . . . . . . . . . . . . . . . . . 43
3.3 Overview of the MCFIFO . . . . . . . . . . . . . . . . . . . . . . . 44
3.4 MCFIFO Architecture [1] . . . . . . . . . . . . . . . . . . . . . . . 45
3.5 Global State Detectors for a 4-place MCFIFO . . . . . . . . . . . . 48
3.6 Global State Detectors Implemented with Dynamic Logic for a 4-place MCFIFO [1] . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.7 Global State Detectors Implemented with Dynamic Logic for a 8-place MCFIFO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.8 MCFIFO Controllers [2] . . . . . . . . . . . . . . . . . . . . . . . . 52
3.9 FIFO Cell's Implementation [2] . . . . . . . . . . . . . . . . . . . . 53

4.1 Data Broadcasting Architecture . . . . . . . . . . . . . . . . . . . . 57
4.2 Synchronous Interfaces of the DBFIFO . . . . . . . . . . . . . . . . 58
4.3 Type-A 4-to-1 Shift Register . . . . . . . . . . . . . . . . . . . . . 59

4.4 Type-B 1-to-4 Shift Register . . . . . . . . . . . . . . . . . . . . . . . . 61
4.5 Data Broadcasting Shared Bus . . . . . . . . . . . . . . . . . . . . . . . 66
4.6 Bus Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.7 Mixed-Clock Shared Bus . . . . . . . . . . . . . . . . . . . . . . . . . . 71

5.1 Synchronous Interfaces of the Shared Buses . . . . . . . . . . . . . . 77
5.2 Example of SoC Modules Configuration . . . . . . . . . . . . . . . . . 79
5.3 Total Cell Area Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.4 Total Dynamic Power Consumption . . . . . . . . . . . . . . . . . . . 81
5.5 Total Cell Leakage Power . . . . . . . . . . . . . . . . . . . . . . . . . 82

B.1 System Configuration Used in DBSB Simulations . . . . . . . . . . . 132
B.2 Waveform of Test Case 1 of DSBS . . . . . . . . . . . . . . . . . . . . 133
B.3 Waveform of Test Case 2 of DSBS . . . . . . . . . . . . . . . . . . . . 134
B.4 Waveform of Test Case 3 of DSBS . . . . . . . . . . . . . . . . . . . . 135
B.5 Waverform of Test Case 1 of MCSB . . . . . . . . . . . . . . . . . . . 136
B.6 Waverform of Test Case 2 of MCSB . . . . . . . . . . . . . . . . . . . 137

# List of Acronyms

| **Acronym** | **Significance** |
|---|---|
| BIST | built-in self-test |
| DFF | d-type flip-flop |
| DBFIFO | data Broadcasting FIFO |
| DBSB | data Broadcasting Share-bus |
| DEC | decoder |
| FIFO | first-in, first-out |
| FPGA | field-programmable gate array |
| FSM | finite state machine |
| IC | integrated circuit |
| IP | intellectual property |
| LSB | least Significant Bit |
| MSB | most Significant Bit |
| MCSB | mixed-clock Shared-bus |
| MCFIFO | mixed-clock FIFO |
| M-MCFIFO | multiple mixed-clock FIFO |
| MSS | metastable state |
| MTBF | mean time between failure |
| MUX | multiplexer |
| MVCML | multiple-valued current mode logic |
| PSSR | parallel-in, serial-out shift register |
| SoC | system-on-a-chip |
| SPSR | serial-in, parallel-out shift register |
| VLSI | very large scale integration |

# Chapter 1

# Introduction

The electronics industry has seen a phenomenal growth over the last two decades, mainly due to the rapid advances in integration technologies to create so called *very large scale integrated* (VLSI) circuits. Today, very powerful computers can be built from several VLSI circuits. As a result, the system size and operation speed are significantly increased. The number of applications of *integrated circuits* (ICs) in high-performance computing, telecommunications, and consumer electronics has been rising quickly. As more complex functions are required in various data processing and telecommunications devices, the need to integrate these functions in a small system/package is also increasing. Recent advances in silicon technology have led to an era where a complete system can be integrated and implemented on a single silicon chip, referred to as a *system-on-a-Chip* (SoC). In an SoC, the feature size of the circuit's building block, a transistor, has decreased into the deep submicron range. Memory modules, logic elements, communications peripherals, and digital & analog modules, formerly attached to the processor at the board-level, are integrated onto the same silicon chip of the processor. The advantages of SoCs are that of combining multiple components into one package leading to cheaper manufacturing of systems, greater design reliability and greater circuit operating speed. SoCs allow devices with smaller size and weight to be built with more features.

They also allow reuse of designs. The complex applications that SoCs can perform along with a narrow market window on certain technologies, favors the movement of converting board-level designs into SoCs.

A typical SoC contains microprocessor cores or *digital signal processing* (DSP) cores, peripherals modules, external interface modules, multimedia modules, networking modules and most importantly, bus and interconnect networks. Some of the typical characteristics of SoCs are:

- Very large number of transistors
- Constructed using third party Intellectual Property (IP) cores
- Mixed technology on the same chip including digital, analog, memory, interconnect, etc.
- Multiple clock frequencies being used on the chip

These characteristics result in greater design flexibility and increase of circuit performance but pose new challenges in SoC design and testing. The buses and interconnect networks are the bridges that connect and enable communications among cores and among external devices in SoCs. Each embedded core has an interface circuitry to interact with. To correctly interface with the core, mixed timing inter-core interconnection is becoming more critical. As the complexity of SoCs grows, the on-chip buses or interconnect architectures start dominating system performance and power consumption. Since the reliability of SoCs has become highly dependent on the fault-free operation of the bus systems, they must be designed to be sufficiently flexible and robust in order to fulfill the wide variety of performance specifications. As a result, design and verification of the mixed-timing bus architecture, and the internal and external bus interface systems are now an essential part of SoC design.

## 1.1 Thesis Organization

In this thesis, two new mixed-clock shared bus architectures are proposed. These architectures address many problems that SoC integrators encounter when designing a high-speed interconnection network for SoC. In Chapter 2, the author discusses the challenge of designing a highly complex deep sub-micron SoC. The problems of designing the interconnection using a globally synchronous approach are stated. Alternative approaches for interfacing highly compacted VLSI circuits with various asynchronous communication methods are then introduced. The advantage and disadvantage of each approach are presented. Chapter 3 explains the design of a mixed-timing interface using Mixed-Clock FIFO, which enables high performance, low latency mixed-timing data transfer between two modules. The design of the proposed mixed-clock shared bus architectures are detailed in Chapter 4. The simulation results and performance of the share bus designs are presented in Chapter 5. Finally, Chapter 6 summarizes and presents the conclusion of the thesis.

[No text]

# Chapter 2

# Mixed-clock Interface for SoC

This chapter contains three sections: Synchronous Designs' Challenges, Asynchronous Design Scheme and Mixed-clock Interface.

Section 2.1, Synchronous Design's Challenges, discusses the issues and limitation of designing a globally synchronous system. Section 2.2, Asynchronous Design Scheme, discusses various asynchronous design styles as alternatives to synchronous design, how subsystems communicate via interface structures that employ asynchronous signaling and data encoding techniques. Section 2.3, Mixed-clock System, introduces a feasible way to design high performance SoCs, discusses how subsystems in mixed-clock SoCs communicate, looks at the concept and architecture of mixed-clock interface using synchronizers, and investigates various synchronization issues.

## 2.1 Synchronous Design's Challenges

Most VLSI designs are built with synchronous logic, where small blocks of combinational logic are separated by synchronously clocked registers. In synchronous systems, sequence and time are connected by means of a global clock signal. The clock is a sequence reference and also a time reference. As a sequence reference, its transitions serve the logical purpose of defining successive instants at which system state changes may occur. As a time reference, the period between clock transitions serves the physical purpose of accounting for circuit and wire delays in the paths from the output of one sequential element to the input of another. The storage dependence of synchronous systems resides entirely within the sequential or clocked storage elements, and the only dynamic characteristic of combinational logic that affects the performance of the system is the circuit's propagation delay. One of the biggest advantages is that synchronous logic made it easy to determine the maximum operating frequency of a design by finding and calculating the longest delay path between registers in a circuit. This often simplifies the design, verification, maintenance and testing process of a system.

### 2.1.1 Global Clock

Synchronous systems use a clock to distinguish a current event in a computation from the previous or the next event. Ideally, this clock should be seen as an identical signal at any point of the system. In order words, it should arrive simultaneously at all clocked elements in the system. Clocked elements in a synchronous system may take any of a variety of forms, including latches and flip-flops, memories and dynamics gates, but they all share the clock as a common time reference. Unfortunately, since the clock is often loaded with the great number of fanout, travels through the longest distances, and operates at the highest speed of any signal in the system, it is extremely difficult to ensure the clock signals to arrive at the same time to all clocked elements. Also, mismatched clock networks' path, and processing and environmental variations introduce differences in the arrival time of the clock from one point to another. The absolute difference between the nominal and the actual interarrival times of a pair of physical clock edges is called clock skew [5]. One consequences of clock skews is the cause of data set-up and/or hold time violation of clocked elements, resulting in synchronization failure between two subsystems in an SoC. Synchronization issues are further discussed in section 2.3.2.1.

Clock skew sources can be classified as systematic, random, drift and jitter. Systematic clock skew is the portion that exits even under nominal condition, due to asymmetric or uneven load to the clock distribution network. Random clock skew is introduced due to processing variations, affecting the thickness and width of wire, transistor channel lengths, threshold voltages or oxide thicknesses, etc. Over time, environmental variations, including temperature and humidity, may develop and eventually cause clock drift. Finally, jitter is a characteristic of the clock generator and can be caused by supply voltage variations and mismatches in the PLL or DLL circuitry. High-frequency environmental variations, like power supply noise and cross-talk between high-speed wires, can also induce jitter into the clock signal.

### 2.1.2 Clock Generation and Distribution

Many synchronous SoCs have their own on-chip clock generators. The clock is normally generated from an electronic oscillator circuit. The period of the clock is typically controlled by a crystal or some other resonant network. Clock generation unit often include a *phase-locked-loop* (PLL) or *delay-locked-loop* (DLL) to regulate the period or phase of the global clock. The global clock is then distributed across the chip through a clock distribution network. The distribution network must deliver the clock across the chip so that it arrives to all clocked elements, via clock buffers, at about the same time. Therefore, that network must be carefully designed to minimize clock skew. Global clock distribution trees can be classified as grids, H-tree, spines, ad-hoc or hybrid [5].

A clock grid is a mesh of horizontal and vertical wires delivering clock signals from clock buffers located in the middle or on the edges. An H-tree is built by recursively constructing fractal H-shaped structures on the vertices of other H-shaped structures. With enough recursion, a large H-tree is eventually constructed that can distribute a clock from the center of a chip to within short distances from all clocked elements of the chip, while maintaining exactly the same wire lengths. A spines structure drives length-matched serpentine wires, from a few rows of clock buffers across a chip, to each small group of clocked elements. The rows of clock buffers can be placed in a way such that the load to each serpentine wire is equal, thus, reducing the systemic skew of the network. An ad-hoc clock distribution network is designed which attempts to equalize wire lengths via manual routing or equalize delays via adding buffers throughout the different parts of a chip. Finally, a hybrid clock tree combines the structure of the H-tree and grid network together in an attempt to compensate clock skews that are introduced when the clock distribution network is constructed with H-trees or grids alone.

### 2.1.3 Synchronous Design Limitations

As the integration scale and the clock frequency on SoCs increase, global synchronization is commonly used to keep a system working by avoiding data read failures. However, global synchronization implemented with delay-matched clock trees has many drawbacks. It needs more metal layers and results in a higher process cost. In many large SoC designs, with millions of transistors, large current surges are necessary to deliver the global clock signal to all clocked elements [6]. The power dissipation of the global clock networks can be up to 40 % of the total power [7]. Also, since SoC designs have not only millions of gates, but also millions of heterogeneous gate structures such as DRAM, SRAM, ROM, flash, digital logic programmable logic and analog circuitry, the job of maintaining the global clock across a chip is becoming more difficult. Significant resources are needed in designing analog devices like PLLs or DLLs to compensate for the propagation delay of local clock buffers and to deal with delay and clock skew reductions.

At the same time, typical clock skew reduction methods need wide metal wires or insertion of delay lines in the clock network [5,8,9]. They usually conflict with power reduction requirements [7]. In addition, the scale of synchronous systems is limited because of timing constraints. It is becoming extraordinarily difficult to find and predict critical path delays in SoCs. Determining the maximum clock frequency is now a more tedious and time consuming assignment than ever. As process technology works further down though the deep sub-micron to the nanometer level, many issues begin to post tremendous problems. Effects such as flicker and shot noise, charge sharing, thermal effects, supply voltage noise and process variations make calculations of delay uncertain and difficult [6]. The physical limits of system scaling and clock frequency will eventually be reached for the future high-performance VLSI design unless global synchronization can be avoided.

## 2.2 Asynchronous Design Scheme

In the early days of digital circuit design, the difference between asynchronous and synchronous circuits was given little regard. Most designers determined that using a clock provided a simpler design methodology. They realized that this added to the power and performance overhead of the chip, but the trade off was worth it. As SoC designs grow larger, designers must grapple with serious global timing problems. The effect of wire loading and timing delays, increased power consumption of the clock distribution network, and the performance penalty associated with supporting on-chip communications, due to increased of clock skew, has encouraged the use of asynchronous communication between subsystems. In this approach, the global clock distribution network in the synchronous scheme is replaced by some variety of special encoding schemes and handshaking protocols to schedule the sequence of events. Asynchronous schemes can be divided into two criteria: completely asynchronous or self-timed system, and system with combinations of locally synchronous modules interfaced with asynchronous and/or other locally synchronous modules. The latter are often called mixed-clock systems.

### 2.2.1 Purely Asynchronous circuits

First, we investigate the concept and structure of a completely asynchronous system to see if it is a practical and feasible solution for the issues caused by globally synchronous design. A totally asynchronous system is a circuit in which the sequencing of its data computations is determined by the data flow and local control signals rather than by clock signals or other global control signals. Control signals for the data transactions are often encoded into the data itself, so that the computation will start when data inputs to a sub-circuit are ready. As soon as the result is computed, the next computation can be initiated. In the absent of a clock signal for time referencing, totally asynchronous circuits employ special handshaking and data encoding schemes to control the sequence of events in the system.

### 2.2.2 Handshaking Protocols

Asynchronous circuits are often modelled as delay-insensitive. A delay-insensitive design assumes that delays in both elements and wires of the circuit are unbounded. With a delay-insensitive model, no matter how long a circuit waits, there is no guarantee that the input will be received. The recipient of data is forced to inform the sender when it has received the data. The sender is required to wait until it gets the completion signal before sending the next data item. There are two widely used signaling scheme, 2-phase and 4-phase signaling, to govern delay-insensitive data transfer. The two forms of signaling conventions are functionally similar in their role of communicating data but differ in details of encoding.

Both signaling schemes can be used to correctly control the flow of data. However, since it costs time and energy to drive a transition onto a wire, the cost will be reduced if fewer transitions are used in an asynchronous signaling convention [10]. For example, in an asynchronous single-rail circuit, there must be at least two transitions for each operation performed by a circuit element. That is, a request (*Req*) has to be asserted to initiate the operation and an acknowledge (*Ack*) is used to indicate the completion of the operation. Suppose that all request and acknowledge wires in the system are initialized to be in the zero state. As long as the request and acknowledge wires are in the same state, it indicates an operation has been completed. If the two wires are in different states, an operation is in progress. Since this signaling scheme takes two signal transitions to complete an operation, it is variously called 2-phase, transition or nonreturn-to-zero (NRZ) signaling. All four states of (*Req*) and (*Ack*) are used in this scheme, as shown in Figure 2.1.

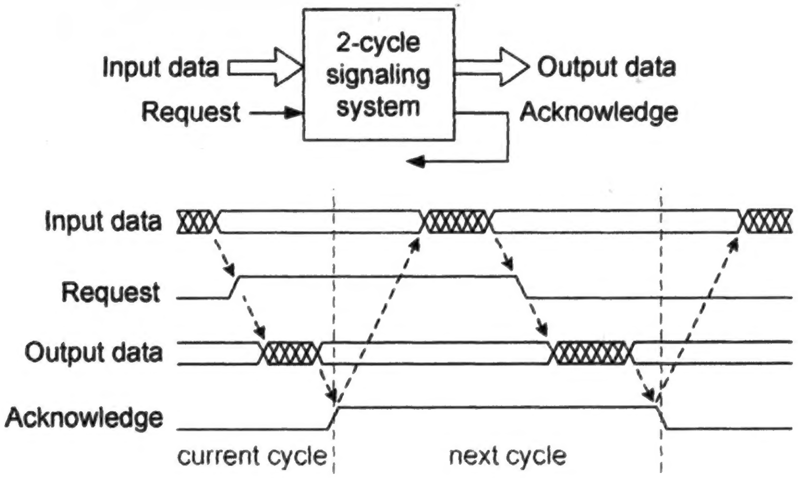


Figure 2.1: Two-phase Signaling

2-phase signaling is fast and energy efficient; however, some extra logic and state information in each element is required to detect each transition of *Req* and *Ack*. An alternative to 2-phase signaling is known as 4-phase, Muller or return-to-zero (RZ) signaling. In 4-phase signaling, a completion of an operation is marked by returning the request and acknowledge wires of an element to the zero state. Therefore two extra signal transition cycles are needed to complete an operation. A form of 4-phase signaling is illustrated in Figure 2.2.

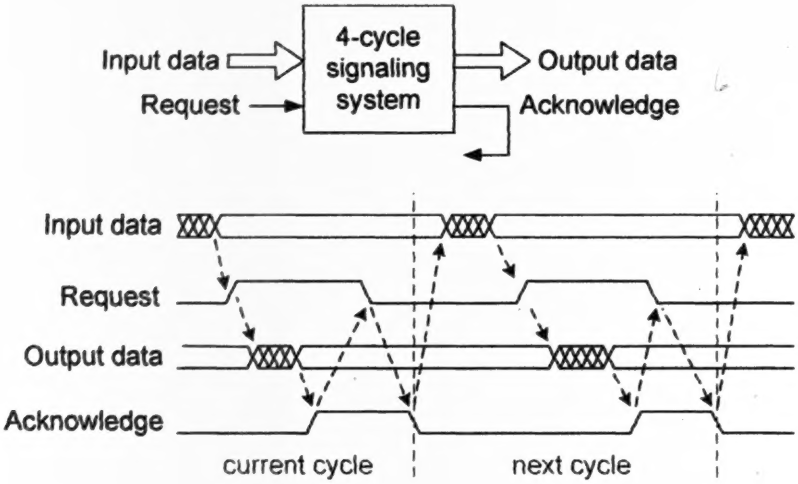

4-cycle signaling system
Input data
Request
Output data
Acknowledge
Input data
Request
Output data
Acknowledge
current cycle
next cycle

Figure 2.2: Four-phase Signaling

The return-to-zero character of 4-phase signaling tends to result in very simple and natural circuit implementations [10]. Therefore, in systems where wire delays are a substantial fraction of the operation time, for example, in long distance communication network, 2-phase signaling scheme should be used. 4-phase signaling is commonly use in local communication, particularly with speed independent elements because of the circuit economy [10].

### 2.2.2.1 Micropipelines

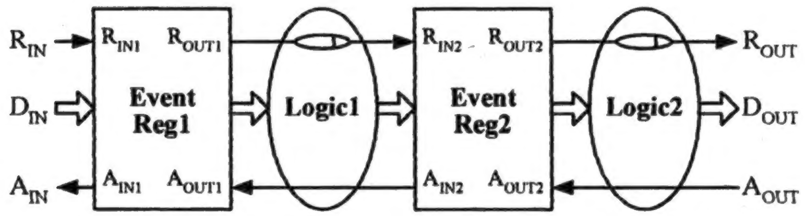
An example of a circuit structure that employs the two signaling scheme is a micropipeline [11, 12] shown in Figure 2.3a. A transaction starts when a request for pushing data onto $D_{IN}$ is made. Once a request event is generated on control line $R_{IN}$, the data is copied into *Event Reg1*, which then signals the event on its $R_{OUT}$

and $A_{IN}$ outputs [11, 13]. In this state, *Event Reg1* holds the data stable until it receives an acknowledge signal on its $A_{OUT}$ input from the event register on the right. Since the design in Figure 2.3a uses a single-rail encoding scheme, a delay element may be needed to be added to each stage of the request line to guarantee that a valid data is made available before the arrival of control signals. The request signal generates by *Event Reg1* is delayed for enough time to allow the data on the outputs of the following logic block, *Logic1*, to be stable. After receiving a request signal on input $R_{IN}$, the second event register *Event Reg2* latches the data, then acknowledges it by signaling an event on its $A_{IN}$ output and generates a request signal on its $R_{OUT}$ output for the next event register. The data processing procedure is repeated for the rest of the micropipeline stages.

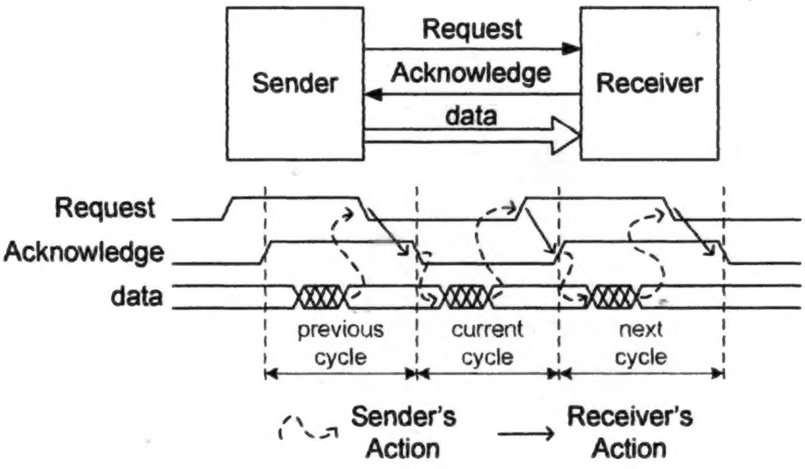
### 2.2.3 Data Encoding

#### 2.2.3.1 Single-rail Encoding

Traditionally, SoC designs use a single-rail encoding scheme to interface between the data sender and data receiver. In asynchronous single-rail encoding, one wire per bit is used to present and transmit data, and an associated request line is used to indicate data validity. The associated channel for the data transfer is called a bundled-data channel [14]. In this scheme, the sender of the data issues a data-valid signal, request, when the data on its output is valid, and the receiver responds by sending a data-release signal, acknowledge, when the data has been processed. The sender has to keep the data on the channel stable and valid at least from the issue of the data-valid until the receipt of the data-release. A sender may take an indefinite amount of time to prepare valid data. After the issue of the data-valid signal, however, it has to keep the data constant. The receiver can then prolong the data-valid period as long as it wants before issuing the data-release signal to mark the end of the period. The micropipeline structure in Figure. 2.3 is an example of a


R_IN
D_IN
A_IN
R_IN1
R_OUT1
Event Reg1
A_IN1
A_OUT1
Logic1
R_IN2
R_OUT2
Event Reg2
A_IN2
A_OUT2
Logic2
R_OUT
D_OUT
A_OUT

(a) Architecture


Sender
Request
Acknowledge
data
Receiver
Request
Acknowledge
data
previous cycle
current cycle
next cycle
Sender's Action
Receiver's Action

(b) Signaling

Figure 2.3: Micropipelines

signal-rail, bundled-data communication channel.

Single-rail circuits provide moderate area efficiency because they use only one wire per bit, plus one additional wire to signal the validity of the data. Single-rail data-path operators such as adders and exclusive ORs can be found in any generic standard-cell library so they are easier to design and verify. Most of the synchronous circuit designs employ a single-rail encoding scheme. Essentially, asynchronous and synchronous single-rail logics are very similar. Instead of using clock edges as triggers for data transfer events, request and acknowledge handshake signals are used to control the flow of data from one stage to another on asynchronous communication networks. Utilizing a signal-rail encoding scheme to construct an asynchronous communication network allows the reuse of synchronous single-rail logic. However, single-rail circuits assume that the delay in the request line is longer than the delay in each of the data wires. The single-rail encoding style bases the handshake upon a matched-delay path (see micropipeline) that suffers from the same timing validation problems as synchronous design. In reality, it can be difficult to match the handshaking signals with appropriate delay models. Therefore careful timing analysis is required to determine the correct amount of delay.

#### 2.2.3.2 Dual-rail Encoding

Many purely asynchronous systems are constructed as delay insensitive circuits. In a delay insensitive circuit, the detection of new data arrival to the circuit's input triggers the start of a computation. To identify the arrival of valid data, protocols are used so that spacers are inserted between valid values. The data is encoded to represent a spacer and the two digital logic states. A common encoding scheme for delay- insensitive computation is dual-rail encoding.

In dual-rail encoding, the data is sent using two wires for each bit of information [15]. The two wires are used to represent data and the request and acknowledge signals are encoded into the data.

The empty or invalid state indicates that no valid data bit is available. The empty state serves as a spacer to separate each valid data bit in a data stream. Suppose a data value, x, is represented using two wires, x_t and x_f. The empty state is represented by {x_t,x_f} = 00. A logic 1 is represented by {x_t,x_f} = 10. A logic 0 is represented by {x_tx_f} = 01. A transition between a logic 0 to spacer or a logic 1 to spacer, and between a spacer to a logic 0 or a spacer to a logic 1 is allowed. However, a transition between a logic 0 to logic 1 or vice versa is prohibited, and the two wires are never in the process of switching at the same time.

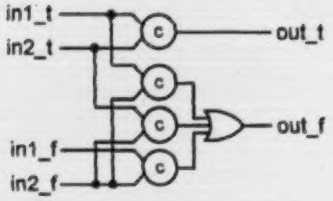
To synthesize a dual-rail delay insensitive circuit for a functional block, a technique called *delay insensitive min-term synthesis* (DIMS) is used [16]. In this technique, the minimal term are formed using Muller C-elements, instead of AND gates. A Muller C-element is a logic circuit with two inputs and one output. When both inputs of a Muller C-element are 1, its output becomes 1. When both inputs are 0, its output becomes 0. Otherwise the output remains unchanged. Table 2.1 illustrates the truth table of the Muller C-element. This synthesis style resembles the traditional sum of product approach but the reduction of the Boolean equations by combining minimal terms is not allowed. This is to assure the combinational circuits do not produce any valid output signals until all input signals are valid, and none of the output signals are empty values until all inputs are spacers. Figure 2.4 and Figure 2.5 illustrate three circuits built using C-elements, NOT and OR gates. Although DIMS does not allow reduction of Boolean equations, multiple logic functions that depend on the same input may share C-elements.

Table 2.1: Muller C-element's Truth Table

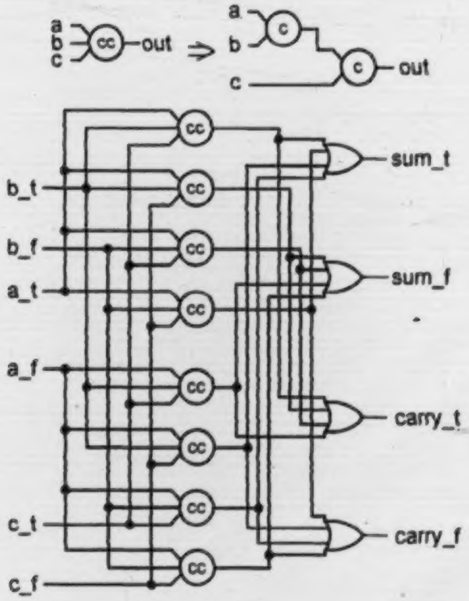
| **Input** | **Output** |
|---|---|
| a = 0, b = 0 | 0 |
| a = 0, b = 1 | unchanged |
| a = 1, b = 0 | unchanged |
| a = 1, b = 1 | 1 |

#### 2.2.3.3 Dual-rail Pipeline

An example of a pipeline structure that utilizes dual-rail encoding is shown in Figure 2.6 [16]. In this pipeline, delay insensitive latches are placed between functional blocks. The latch holds the value of the input data and sends it to the successive circuits when they are ready to receive. Each latch is controlled by acknowledge


in1_t
in2_t
in1_f
in2_f
c
out_t
out_f

(a) Dual-rail AND


a
b
c
cc
out
⇒
a
b
c
c
c
out
b_t
b_f
a_t
a_f
c_t
c_f
cc
sum_t
sum_f
carry_t
carry_f

(b) Dual-rail Adder

Figure 2.4: Dual-rail Circuit Implemented using DIMS

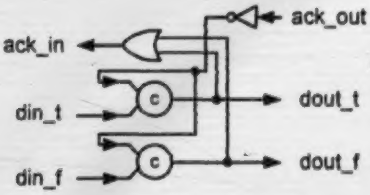

ack_out
ack_in
din_t
din_f
c
dout_t
dout_f

Figure 2.5: Asynchronous Latch

signals from its succeeding latches. If a latch receives $A_{IN}$ as low, it is allowed to load and hold a valid data value and outputs $A_{IN}$ as high to its previous latch. If the $A_{OUT}$ is high, the latch can load and hold a spacer and outputs $A_{IN}$ as low to its previous latch. A delay insensitive ring can also be formed with at least three latches, by connecting the output of the last stage to the input of the first. Such ring is often used to perform iterative computations.

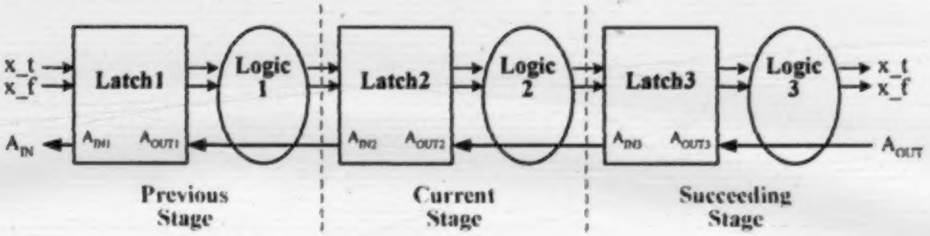


Figure 2.6: Dual-rail Pipeline

#### 2.2.3.4 1-of-n and m-of-n Encoding

The concept of dual-rail encoding can be extended to create a more power efficient, delay-insensitive encoding scheme known as 1-of-n encoding. In 1-of-n encoding, a symbol, which is a 2-bit code (01 = logic 1, 10 = logic 0, 00 = spacer, etc), is transmitted on one of the group of n wires [17]. The most common form is a 1-of-4 or one-hot encoding [18], where a symbol is transmitted on one of four wires. Two bits of information are thus carried over four wires, giving the same resource requirements as a dual-rail encoding scheme. However, since a 1-of-4 can convey 2 bits of information in one data transfer cycle, it is more power efficient than dual-rail encoding. A superset of 1-of-n encoding is called m-of-n encoding [19], where more than one, or m symbols, can be transmitted on a group of n wires.

#### 2.2.3.5 Color Coding

Dual-rail encoding is also known as 1-color 4-phase coding. The term "1-color" refers to the number of valid data representation and "4-phase" refers to the number of communication actions between a data sender and receiver. In dual-rail encoding a spacer must be transmitted in between valid data. To eliminate the timing overhead due to the insertion of spacers, other color coding convention can be utilized.

Two additional color coding schemes, 2-color 2-phase encoding and 2-color 1 phase dual-rail encoding, are introduced in [20]. In 2-color 2-phase encoding, valid data are encoded into two colors, EVEN or ODD. Each successive data bits in a data stream must have the opposite color. Color of the data item is indicated by the request and the acknowledge signals. If the sender receives the color of the acknowledge signal as ODD, it sends data with an EVEN color. The receiver detects EVEN data and responds to the sender by changing the color of the acknowledge signal to EVEN. 2-color 2-phase encoding requires an additional wire for the acknowledge signal. However, by using *multiple-valued current mode logic* (MVCML), the color and data information can be encoded onto two wires. This encoding method is called 2-color 1-phase dual-rail encoding. The use of MVCML is beyond the scope of this work and, therefore, will not be discussed further in this thesis.

Both single-rail and dual-rail encoding schemes are commonly used in asynchronous circuit design, and there are tradeoffs between each. The benefit of the dual-rail approach is that it allows for data validity to be indicated by the data itself. The handshaking signals are generated upon the arrival of data so it is not necessary to perform delay matching on the request line. Unfortunately, the realizations of dual-rail or m-of-n implementations for asynchronous handshake circuits are rather area inefficient. For each bit two or more wires are used in the encoding compared to one wire for synchronous circuits. Since each wire has to be driven by some cells and hence requires additional transistors and circuit area (twice or more the area that of an equivalent synchronous are need to implement a m-of-n circuit) [14]. An increased circuit area implies longer wires which consume more power and potentially lowers speed performance. Furthermore, double-rail combinational logics are generally implemented using dedicated dual-rail cells, not normally available in a standard cell library. Finally, dual-rail designs often prohibit the reuse of existing IP cores, which mainly consist of synchronous single-rail logics. Therefore, it is difficult to design complex systems with such data encoding schemes.

### 2.2.4 Self-timed Systems

In a self-timed system, sequence and time are connected in the interior of circuit parts called self-timed elements [10]. Self-timed elements are required to be contained over equipotential regions, where the propagation delay on any wire within the region is assumed to be small compared to the switching or signal transition delay. The signal transitions at the terminal of a self-timed element may occur in a certain order, such that, the sequential operation of a self-timed system is insensitive to element and wiring delays. To ensure correct operation of data transfer within the self-timed system, a handshaking protocol must be used to provide control on the asynchronous interconnection. Two self-timed signaling conventions, equipotential and delay insensitive signaling, are used as handshaking schemes.

Over the small equipotential regions, the related signals that are carried on wires may be treated as identical at all points on the wires. Equipotential signaling conventions imply the assumption that within a region, wires sustain negligible delay so that any relations produced in the region holds everywhere [10]. In this case, open-loop orderings at an element's output can assure that those orderings are preserved at the inputs of other elements. Therefore, a bundled data convention can be used as the signaling scheme, in which data-validity information costs only one single wire. However, wires between equipotential regions are assumed to have a sustained arbitrary delay. Delay insensitive signaling is used for communication between elements that are not in the same equipotential region, i.e. dual-rail or m-of-n encoding scheme is used.

### 2.2.5 Disadvantage of Asynchronous Systems

In general, asynchronous systems are most suitable for simple functions with few inputs and outputs because the design complexity of these circuits increase drastically with the number of inputs and outputs [21]. On the other hand, self-timed systems are required to be constructed with elements that are contained over equipotential regions, in an attempt to simplify the complexity of the handshaking scheme. However, as CMOS feature size shrinks, the rate at which the size of the equipotential region shrinks is faster than that of the circuit sizes. Consequently, less devices can be accommodated within each equipotential region and imposing a serious performance limitation for self-timed systems [22]. Eventually, it is difficult to distinguish self-timed systems from purely asynchronous circuits as they both share the same disadvantages. The entire platform needs to be designed asynchronously to achieve a completely asynchronous SoC design. In other words, interface hardware, memory, and supporting glue logic will all have to be asynchronous. The large circuit overhead induces more difficulties to design highly complex SoCs as purely asynchronous or self-timed system.

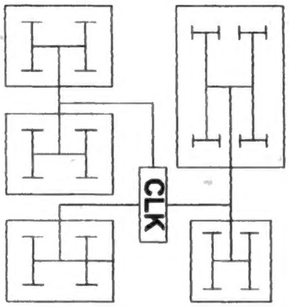
Meanwhile, design support comes in as another major issue. Existing design tools are barely adequate for synchronous designs, let alone asynchronous design, where tools are virtually nonexistent. Using current asynchronous design tools, like Petrify, Balsa and proprietary tool likes Tangram, require a significant re-education of designers, and their features are far behind synchronous commercial tools [23].

Many asynchronous designs are achieved using modified synchronous tools. *Computer Aided Design* (CAD) tools are deceived into thinking that the designs are synchronous. The shortcomings of this are that logic design verification, timing checks, and race condition checks become very difficult.

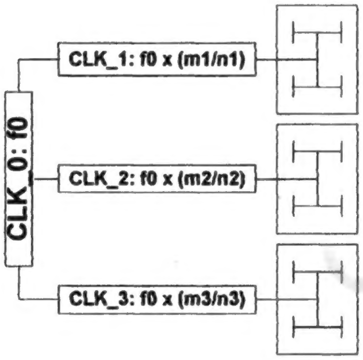
Debugging complex asynchronous logic is also extremely difficult. In a synchronous design, a lower clock frequency can be used to detect and diagnose certain faults and determine fixes. However, in an asynchronous design, there is no such clock and the logic must be debugged at full speed to identify a failing logic path. A false transition on an output from one core can cause the next one to operate on meaningless results. Designers often need to base their judgements on circumstantial evidence to find and correct design faults, which increases time-to-market and development costs. The lack of testing architectures for asynchronous SoCs also posts a tremendous hurdle for designers. Once again, designers may need to find ways to fool the testers into thinking that the logic is synchronous, in order to make conventional *automated test equipment* (ATE) to perform asynchronous design testing. Consequently, it is not desirable and often not feasible to design purely asynchronous SoCs.

## 2.3 Mixed-clock System

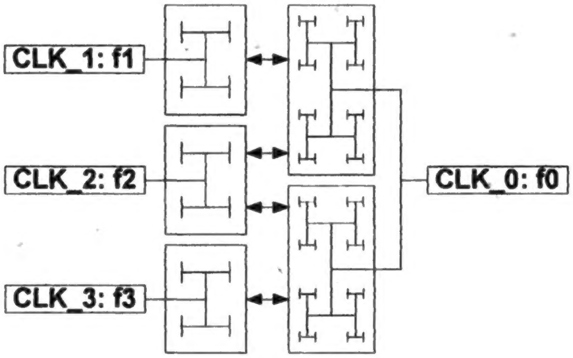
The issues of synchronous SoC design appear in terms of serious global timing problems, effects of wire loading and timing delays, and performance reduction associated with supporting on-chip communications between different clock domains. On the other hand, the overwhelming complexity of designing and testing a purely asynchronous SoC platform also prohibits most designers from adapting to such a design approach. The answer to the design dilemmas lies in combining the understood and predictable synchronous methodologies with asynchronous design



(a) Single Clock Frequency


CLK_0: f0
CLK_1: f0 x (m1/n1)
CLK_2: f0 x (m2/n2)
CLK_3: f0 x (m3/n3)

(b) Rational Clock Frequencies


CLK_1: f1
CLK_2: f2
CLK_3: f3
CLK_0: f0

(c) Multiple Clock Frequencies

Figure 2.7: Various Types of Multiple Clock Domains

techniques. For example, an asynchronous circuit can be constructed with locally synchronous subsystems with independent clocks but communicate asynchronously on the system level. External asynchronous signals to these subsystems are made internal via synchronizers or other asynchronous interface structures. The local clocks can have the same frequency with arbitrary phases or, all together, different frequencies (Figure 2.7).

In particular, mesochronous is used to describe the clocks with the same frequency but have different phases [7]. In a mesochronous system, clock distribution is integrated in the buses, called strobe signals [24], going into each modules, as shown in Figure 2.8. Each block can use the strobe signal as its own local clock. In addition, a plesiochronous interface is characterized as a design that has multiple clock domains with independent clocks which are closely matched in frequency [25].

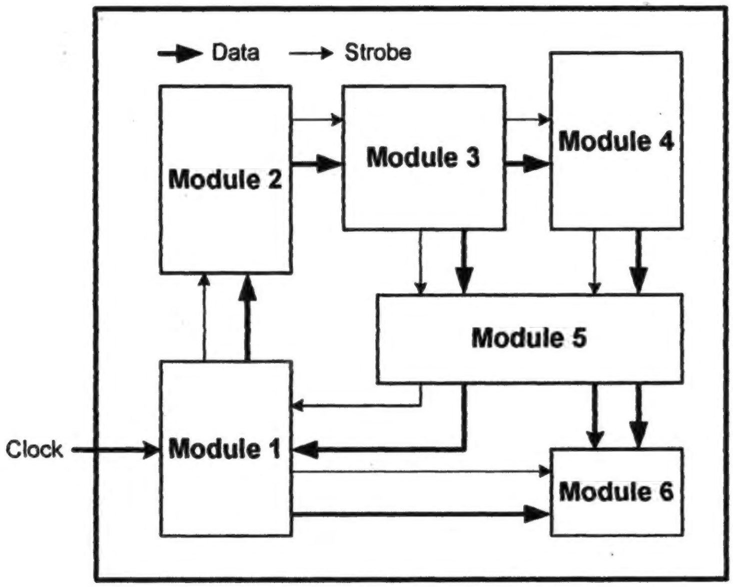


Figure 2.8: Mesochronous Clocking

Since there is no global time reference in mixed-clock systems, the local clocks can be operated at a rate determined locally by the elements and wiring delays of the subsystems. Therefore, the local clock rate tends to be determined by the maximum delay of the local module, instead of the worst-case delay of the overall system. The mixed-clock approach for building large SoCs is gaining more attention as a promising alternative to global clocking schemes. By adding the asynchronous interfaces to locally synchronous modules, the interior of a specific module is isolated from the interfaces. Each synchronous module can employ its own clock and power supply without affecting other modules. Consequently, the problem of clock distribution, clock slew and clock power consumption due to large chips can be mitigated. The individual subsystem can be designed, verified and tested using traditional methodologies associated with synchronous design. This approach also simplifies SoC designs by promoting the reuse of modules.

Since local modules have independent clocks, there is no guarantee that data transfer will be error free. Information communicated from one module to another must be re-synchronized to the receiver's clock. In the following sections of this chapter, different methodologies for interfacing the locally synchronous subsystems are discussed.

### 2.3.1 Mixed-clock Interface

There are two categories of mixed-clock interfaces which can be used to enable communication among submodules: clock synchronization systems and data synchronization systems.

In a clock synchronization system, the submodules communicate through the synchronization of the sender's and receiver's clocks. They can either have *pausible clocks* or rely on exact or nearly exact frequency matching of the clocks to achieve that synchronization.

#### 2.3.1.1 Globally-Asynchronous Locally-Synchronous (GALS) Systems

The term, GALS, was originally proposed by Chapiro [26] in 1984. GALS design use pausible clocks to allow synchronous modules to communicate using asynchronous protocols. Pausible clocks are ring oscillators that can be halted. In a GALS system, each synchronous module has its own clock generator which consists of a ring oscillator and a handshaking stage. When a valid data is ready, it can be transferred from the sender to the receiver during the clock-paused period [27]. First, the sender forwards a request to the receiver and stalls its clock until it receives an acknowledgement. The receiver detects the request and prepares to receive the data by also stopping its local clock. Once the receiving module obtains the data, it sends an acknowledgement to the sender to trigger the restarting of its clock. Finally, the acknowledge signal is set back to low again, and the receiver's clocks restarts and normal operations of the two modules resume. Several approaches have been proposed to stop and restart the clock [27–30]. These approaches all include a synchronous element in their ring oscillators, called a mutual-exclusion element, that can suspend the clock at one time and generate the handshaking signal at another. Figure 2.9 illustrates the structure of a pausible clock generator with the mutual-exclusion element. In GALS systems, an asynchronous wrapper similar to the one shown in Figure 2.10, are implemented as the interface between different clock domains.

The ring oscillators pause when the asynchronous environment wants to communicate with the module. In each cycle, the mutual-exclusion element acts as the arbiter to decide whether the local clock signal or the environment's handshaking signal may proceed. The fact that either the clock or the handshake may proceed implies that the input and output operations are mutually exclusive. A communication between two clock domains starts from one clock domain to the handshake

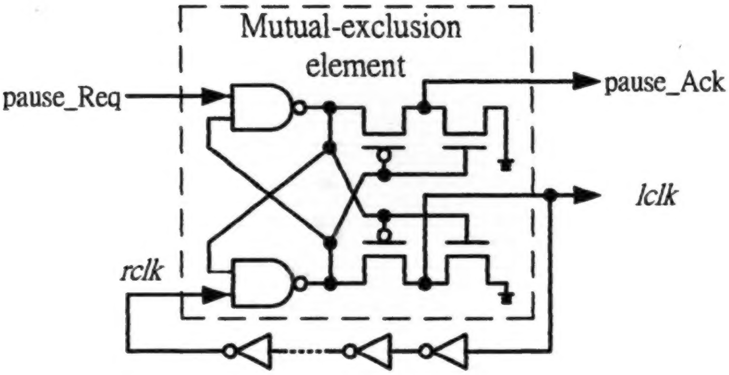

Mutual-exclusion element
pause_Req
pause_Ack
lclk
rclk

Figure 2.9: Pausible Clock Generator

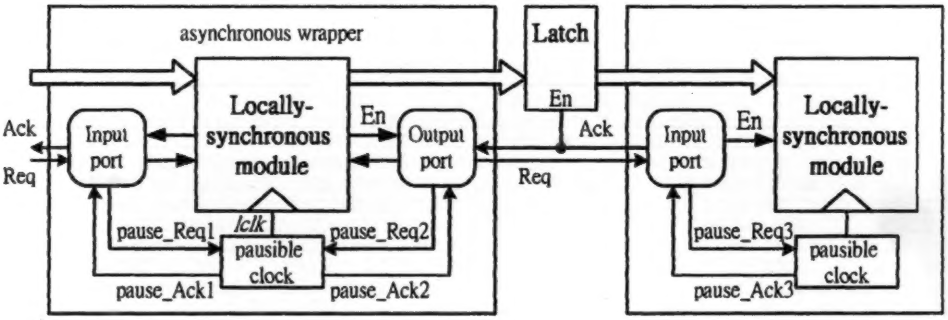

asynchronous wrapper
Latch
En
Ack
Req
Input port
Locally-synchronous module
Output port
En
lclk
pause_Req1
pause_Req2
pause_Ack1
pause_Ack2
pausible clock
Ack
Req
Input port
En
Locally-synchronous module
pause_Req3
pause_Ack3
pausible clock

Figure 2.10: Asynchronous Wrapper and Configuration of a Data Channel in GALS

interface and then from the handshake interface to the other clock domain. Therefore, one clock is paused first and then the other. However, the fact that pausible clock designs must resolve metastability in the arbiter makes their worst-case cycle time unpredictable. Also, GALS design makes minimal assumptions about clock stability. For example, the performance of synchronous systems can be degraded significantly by the clock's stability and low-jitter [25]. Clock pausing inevitably exacerbates jitters, since after suspending a clock, the first edge through the ring oscillator and clock buffer will propagate slower than subsequent edges [31]. In general, designs using this approach suffer higher throughput penalties due to the constant local clock adjustment. The absence of long term predictability and the increase of jitter are major problems encountered when converting a synchronous system into a GALS design. These problems prevent GALS the approach from being used in many large deep sub-micron SoC designs.

#### 2.3.1.2 Synchronization by Frequency Matching

In [25], Chakraborty and Greenstreet proposed a family of interface circuits that mediate between mixed-lock domains by exact or nearly exact frequency matching of the clocks. The basic design consists of a single-stage *first-in first-out* FIFO and a latch controller, as shown in Figure 2.11. This basic design can handle interfaces between two clock domains which operate at exactly the same frequency but have arbitrarily clock phase, i.e. a mesochronous system. The latch controller takes $\Phi_T$ and $\Phi_R$ as inputs and generates $\Phi_X$ that satisfies the set-up and hold requirements of latch-X and latch-R. Clock timing for the FIFO is shown in Figure 2.12, where, $t_s$, $t_h$ and $t_{prop}$ denote the set-up time, hold time and propagation delay of the latches, respectively. Assuming the latches are all positive edge triggered, the set-up and hold time requirements for latch-X are satisfied if the rising edge of $\Phi_X$ occurs at least $t_s$ + $t_{prop}$ after the previous $\Phi_T$, and at least $t_h$ - $t_{prop}$ before the next

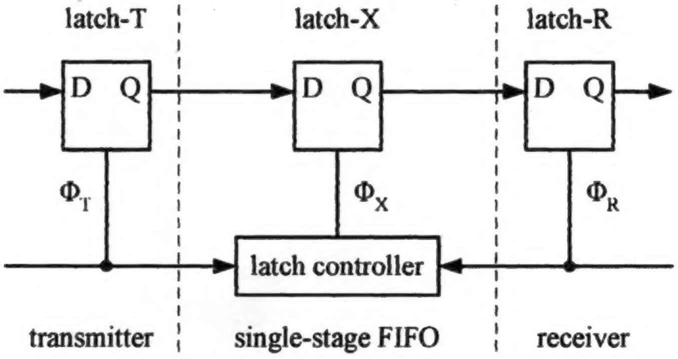


Figure 2.11: Single-Stage Frequency Matching FIFO

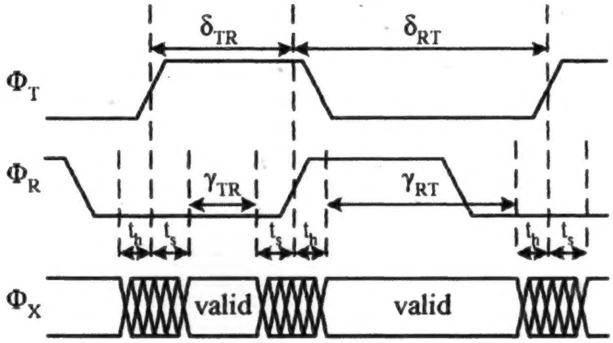


Figure 2.12: Timing Diagram of Single-Stage Frequency Matching FIFO

$\Phi_T$. On the other hand, the timing requirements for latch-R are satisfied if the rising edges of $\Phi_X$ occur at least $t_h$ - $t_{prop}$ after the previous $\Phi_R$, and at least $t_s$ + $t_{prop}$ before the next rising edge of $\Phi_R$. Therefore, there are two windows of opportunity for a valid $\Phi_X$ to be generated. They are denoted by the two valid regions shown in Figure 2.12. Also, $\delta_{TR}$ and $\delta_R T$ in the figure denotes respectively the time from the rising edge of $\Phi_T$ to the next rising edge of $\Phi_R$ and from the rising edge of $\Phi_R$ to the next rising edge of $\Phi_T$. If the width of the two windows are indicated by $\gamma_{TR}$ and $\gamma_{RT}$, respectively, and $P$ represents the clock period. Then the following equations can be obtained.

$$\gamma_{TR} = \delta_{TR} - 2(t_s + t_{prop}) \tag{2.1}$$

$$\gamma_{RT} = \delta_{RT} - 2(t_h - t_{prop}) \tag{2.2}$$

$$\gamma_{TR} + \gamma_{RT} = \delta_{TR} + \delta_{RT} - 2(t_s + t_h) \tag{2.3}$$

$$\gamma_{TR} + \gamma_{RT} = P - 2(t_s + t_h) \tag{2.4}$$

$$\max(\gamma_{TR}, \gamma_{RT}) \geq P/2 - (t_s + t_h) \tag{2.5}$$

Therefore, if the clock period is greater than $2(t_s{+}t_h)$, the latch controller can generate a $\Phi_X$ event, either after the rising edge of $\Phi_T$ (for $\gamma_{TR} > 0$) or after the rising edge of $\Phi_R$ (for $\gamma_{RT} > 0$), ensuring proper operation of the interface. Since the timing of the design is determined by the rising edges of the clocks, proper delay added into the latch controller to ensure the set-up and hold time requirements for the latch-X and latch-R are met. The implementation of the latch controller is shown in Figure 2.13. The single-stage FIFO is shown to handle clock jitter while introducing no additional latency penalties. Based on this basic design, two extensions of the single-stage FIFO, which handle rational clock frequency multiples and plesiochronous system, were proposed in [25]. The two designs demonstrate an average latency of only half a clock cycle and a worst-case latency of two clock cycles. In addition, a FIFO interface that can handle continuous data transfers between a data sender and receiver were briefly discussed. Figure 2.14 shows the FIFO interface implementation presented in [25].

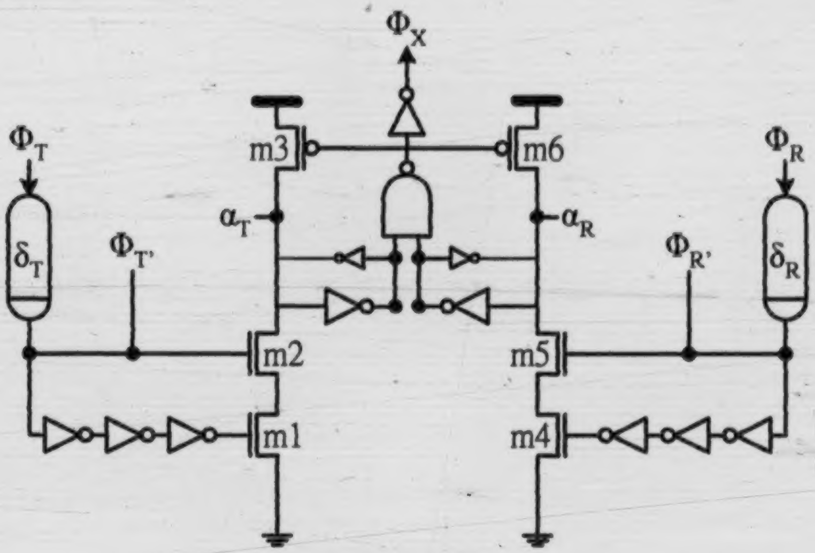


Figure 2.13: Latch Controller

There are several tradeoffs between this approach when comparing with others. First, this FIFO design is more restrictive since the basic design is limited to mesochronous clock domains and to the two special cases of mixed-clock domains

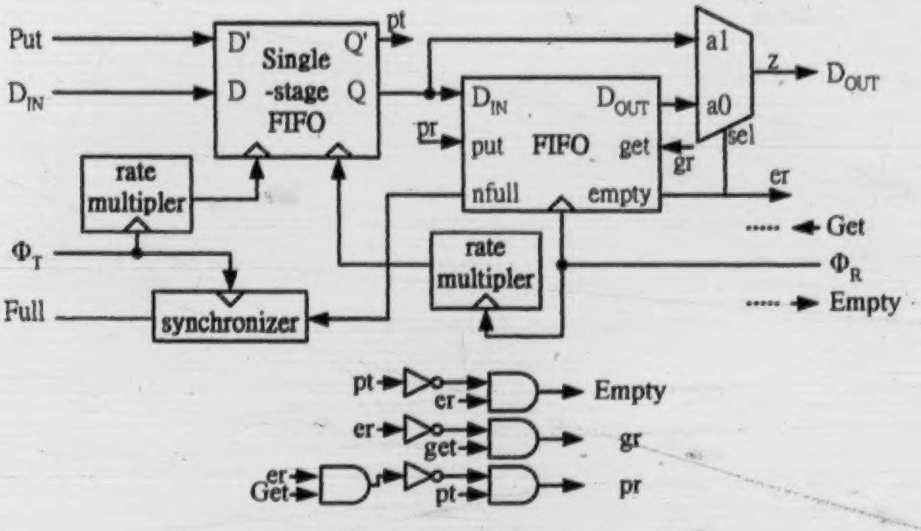

Put
$D_{IN}$
D'
D
Single
-stage
FIFO
Q'
Q
pt
rate
multipler
$\Phi_T$
Full
synchronizer
$D_{IN}$
$D_{OUT}$
pr
put
FIFO
get
nfull
empty
gr
a1
a0
sel
z
$D_{OUT}$
er
Get
$\Phi_R$
Empty
rate
multipler
pt
er
Empty
er
get
gr
er
Get
pt
pr

Figure 2.14: Implementation of a Frequency Matching FIFO Interface

(rational clock domains and plesiochronous clock domains). In contrast, most mixed-clock FIFO designs can handle arbitrary mixed-clock interfaces [1, 32, 33]. Furthermore, the frequency matching approach requires thorough timing analysis of the subsystems in order to calculate set-up and hold times, and the propagation delay of the clock signals. This makes the approach technology dependent and difficult to design. Finally, for proper operation, the interface must be initialized with a special procedure [25], where the latch controller may need to operate with a slight slow-down during this initialization and brought to full speed under normal operation. These procedures require extra control sequences to be executed and further reduce the robustness of the interface.

#### 2.3.1.3 Data Synchronization System

An alternative approach attempts to synchronize data items and/or control signals with the receiver without interfering with its clock. In a data synchronization system, the subsystems have free running, stable clocks in each domain, and make minimal assumptions about the timing relationships between the clocks. The data being transferred is synchronized from one clock domain to the other. The benefit of such approach is that synchronizers can be designed using a conversational synchronous design method. The designer can therefore look at the design from a system-level perspective. Unlike micropipeline and the frequency matching interface in [25], delay elements are not required for proper operation of the interface. Thus, simulation and synthesis of the design utilizing synchronizers are much easier to realize. In fact, a few asynchronous communication schemes which employ synchronizers [1,34] make minimal assumptions about the transistor technology be used to implement the design. As a result, the design efficiency of SoCs can be increased.

### 2.3.2 Synchronizer and Mixed-Clock FIFO

According to Dally [32], a synchronizer is a circuit that samples an asynchronous signal and outputs a version of the signal which is synchronized with the sample clock. If an input signal and a time reference or a voltage reference were given to a system, a decision must be made by the synchronizer to select whether the transition of the input signal happens before or after the time reference, or whether the input voltage is higher or lower than the voltage references. In other words, a synchronizer is used to provide the system time to resolve a *metastable state* (MSS). The most common and simplest method for transferring data between two mutually-asynchronous clock domains is to use a two flip-flop synchronizer.

Figure 2.15 illustrates a basic two flip-flop synchronizer design. In this example, the request and acknowledge lines are synchronized by the receiver and the sender, respectively, and bundled-data convention is employed.

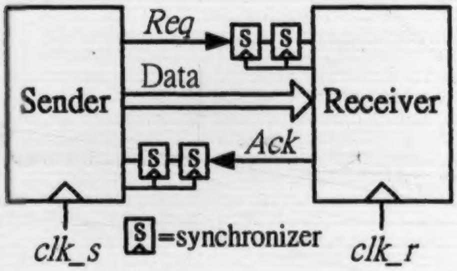


Figure 2.15: Two Flip-flip Synchronizer

#### 2.3.2.1 Synchronization Issues

The major concern about mixed-clock systems employing the synchronizer is that there is no complete certainty on how reliable the synchronization is. The reason is that synchronizing elements are bistable, i.e. the output stages of the elements are constantly changing from 1 to 0, or 0 to 1, and may occasionally enter a MSS. Metastability is the ability of a non-equilibrium state to persist for a long period of time in a sequential element. In digital circuit, an input signal to an element that neither satisfies the condition of being a logic 1 nor 0 is said to be in a non-equilibrium state. When sampling a changing input signal with the clock, the order of events determines what will be sampled [32]. The smaller the time difference between events, the longer it takes to determine which event happens first. The inputs can force the element to enter a MSS when two input events come very close together, requiring a longer time than allotted to decide the outcome of the sampling, and results in synchronization failure. Figure 2.16 illustrates a MSS of a data transaction from one subsystem to another. If the incoming data rate is slow enough so that the chosen flip-flop has enough time to resolve the metastable condition before the next clock edge arrives, then one synchronizer stage implemented using a flip-flop is sufficient for the synchronization.

The period of time in which the output of the flip-flop may go metastable is called the *metastability window*, W, and is defined as the sum of the set-up and hold times of the clock signal driving the flip-flop. A data read from the synchronizing flip-flop may fail if input data arrives during W. In that case, a metastable output will stay undefined for a longer amount of time, increasing the propagation delay. It can also cause the output to oscillate or move to the wrong state. Adding more synchronizers may allow a longer time, $T_r$ to resolve the metastable condition and may reduce the chances of the output being metastable.

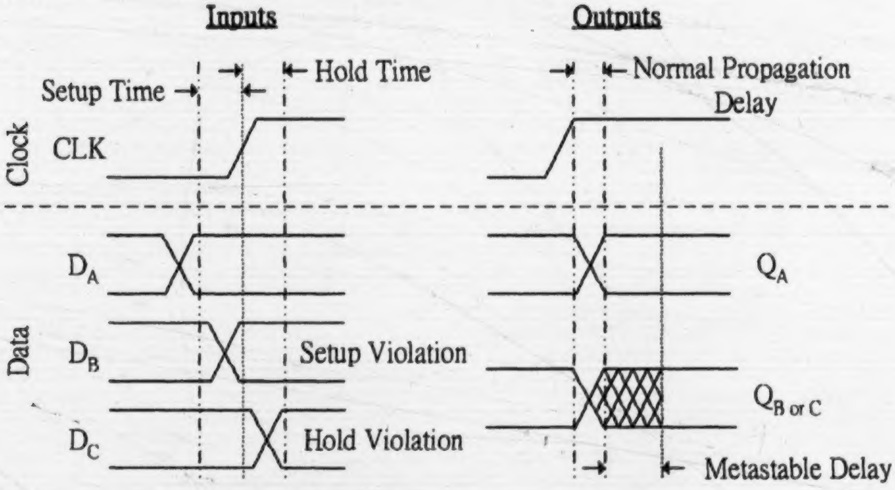

Inputs
Outputs
Setup Time
Hold Time
Normal Propagation Delay
Clock
CLK
Data
$D_A$
$D_B$
$D_C$
Setup Violation
Hold Violation
$Q_A$
$Q_{B \text{ or } C}$
Metastable Delay

Figure 2.16: Metastability in Data Transaction

The synchronizer reliability is given by its *mean time between failure* (MTBF), which is the failure rate of a synchronizer due to the metastable condition [34]. For a two flip-flop synchronizer, assuming the two flip-flops are identical, its MTBF is given by:

$$MTBF = \frac{e^{\frac{T_r}{\tau}}}{f_C \times f_D \times W}$$

where $f_C$ is the frequency of the receiver's clock, $f_D$ is the rate of the data pushing across the clock domain boundary. Assuming that the sender may not send data every clock cycle, $f_D$ is usually not equal to the sender's clock frequency. $T_r$ is the time available to resolve the metastable condition, namely the time separation between two clocked inputs to the synchronizer. $W$ is the metastability window. $\tau$ is the resolve time constant or the characteristic time for the transient of the flip-flop. For an SoC design implemented in $0.18um$ CMOS technology, a typical value for $W$ is about $50ps$ and $\tau$ is about $10ps$ [34]. To decrease the probability of synchronization failure, the factor $T_r/\tau$ has to be maximized. Therefore, careful considerations have to be made in synchronizer design to reduce the chance of it entering a MSS.

A bistable element in a properly designed self-contained synchronous system never has the opportunity to reach a metastable condition since satisfaction of the timing constraints assures such condition would not happen. However, in a mixed-clock system, it is possible for a synchronizer to hover between two well-defined stable states (0 and 1) because it is not properly triggered by its clock. MSS occurs under the conditions in which synchronizers must operate in a mixed-clock system. There is no upper bound for the time the bistable element may remain in this metastable condition. If a metastable state is not resolved soon enough, it may cause a data read failure of the receiver.

In addition, two other issues can arise to cause synchronization failure. If the incoming data rate exceeds that which can be accommodated by the synchronizer, the synchronizing buffer overflows, resulting in a loss of information. On the other hand, a synchronizer underflow is a condition in which the data rate is too low and an attempt is made to access the synchronizer before a data item arrives and an empty buffer is read. Appropriate measures have to be employed to reduce the probability of such faults occurring. Fortunately, if a mixed-clock FIFO is designed carefully, these issues can be minimized.

### 2.3.2.2 Latency

Since synchronizers have to deal with metastability and overflow/underflow states, the designer has to trade off between reliability and low latency. It is understood that attempts to resolve metastability increases the latency of the overall system, i.e. a longer $T_r$ improves reliability but also increases latency. Therefore, a low-latency interface design for mixed-timing systems is particularly desirable.

In the example shown in Figure 2.17, the two sets of synchronizers are connected to two finite state machines (FSM). The protocol that the two FSMs implement is shown in Figure 2.18. A valid data send request signal, $valid$, enables the latching of data into $REG_S$ and triggers the *SFSM* to generate $Req = 1$. $R2$ enables $REG_R$ to latch the data and starts the *RFSM* to send $Ack$ back to the sender. Assuming that $T_r$ equals one clock cycle of the receiver's clock, if a request is sent to the first flip-flop, three possible outcomes may happen. In the first scenario, the rising edge of the request is sampled to high by the first flip-flop. The $R2$ goes high on cycle 2, and since $T_r$ is set to be one clock cycle, the receiver waits a full cycle before latching the data into $REG_R$ at the beginning of cycle 3.

In the second scenario, the rising edge of request is initially sampled as low by the first synchronizer and is sampled as high in cycle 2 if $Ack$ stays low. The output of second flip-flop goes high in cycle 3, and data is latched by the receiver in cycle 4. In the last scenario, the first flip-flop goes into an MSS. If the MTBF is considerably high, then $1 - e^{(\frac{T_r}{\tau})}$ is approximately equal 1. The first synchronizer has exited metastability and arbitrarily settles to either high or low before the beginning of cycle 2. If $R1$ is high, $R2$ goes high in cycle 2, and data is read by the receiver in cycle 3. If $R1$ is low, $R2$ goes high in cycle 3, and data is latched by $REG_R$ in cycle 4. Therefore the worse case latency, calculated from the moment a data item is ready to be sent to the time the receiver latches that data item, is four clock cycles.

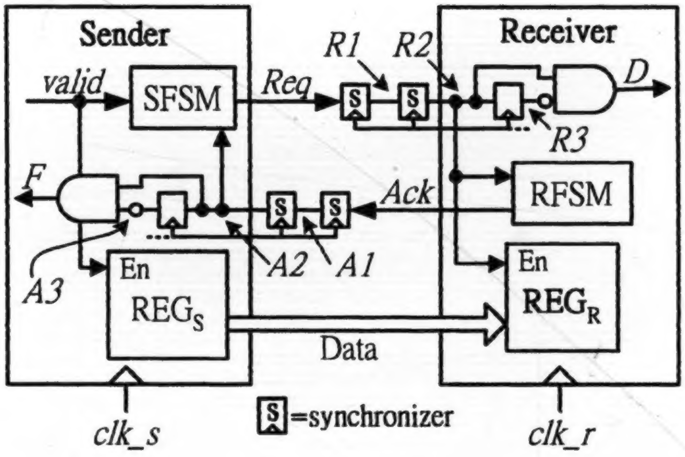

Sender
valid
SFSM
F
A3
En
$REG_S$
clk_s
Req
R1
R2
Ack
A2
A1
Data
S =synchronizer
Receiver
D
R3
RFSM
En
$REG_R$
clk_r

Figure 2.17: Implementation of a Two Flip-flop Synchronizer Data Channel

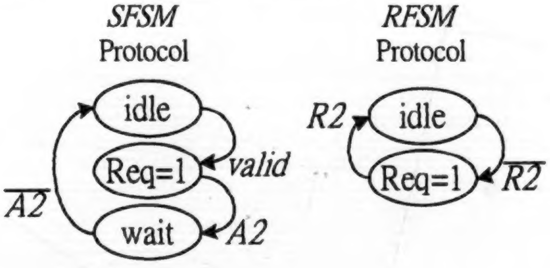

SFSM
Protocol
idle
Req=1
wait
valid
A2
$\overline{A2}$
RFSM
Protocol
R2
idle
Req=1
$\overline{R2}$

Figure 2.18: FSM Protocol

There has been a large body of related work on using synchronizers. Some have focused their efforts to reduce the latency penalties imposed by the synchronizers. Seizovic [33] proposed an interface through a synchronization pipeline. Communication between each stages of the pipeline follows a two-phase protocol. However, the latency of his design is proportional to the number of FIFO stages in the pipeline. His synchronizer design is based on the mutual-exclusion element which is expensive to implement.

Dally [32] and Pham [35] also proposed mixed-clock FIFO designs. In [32], Dally suggested to use $2n$ synchronizer to synchronize the addresses for the head and tail of the FIFO, where $n$ is the number of data items *enqueued* in the FIFO. This gives his design a greater area overhead. Also, his design attempts to synchronize the head and tail on every clock, which significantly reduces the overall throughput of the FIFO.

In [35], Pham used a FIFO to provide communication between two mixed-clock subsystems at low frequencies. However, it fails to provide synchronization of the control signals, namely request and acknowledge. The lack of synchronization of the global control signal for the asynchronous communication makes the design unsuitable for use at higher clock frequencies, where the failure rate may increase significantly to an unacceptable level.

Chelcea and Nowick [1, 2] further optimized the FIFO approach by noting that synchronizations of the global control signals are only needed when the FIFO approaches full or empty. Therefore, if successive data items are sent on a continuous basis, the synchronization overhead is minimal and the asynchronous communication enjoys a very low-latency. Their design creates a highly robust, high performance and low-latency interface for mixed-clock systems.

All of the research works discussed above accommodates asynchronous point-to-point communication for synchronous modules. The proposed mixed-clock shared bus design utilizes Chelcea's Mixed-Clock FIFO and optimizes it further so that it can provide multi-point, mixed-clock interconnection for SoC modules.

## 2.4 Summary

In summary, this chapter reviews challenges of designing a highly complex deep sub-micron SoC. In particular, the problems of designing such system using globally synchronous approach are discussed. Alternative approaches for designing highly compacted VLSI circuits which involve abandoning the use of global clock and control signals are investigated. We look at different ways for the subsystems to communicate asynchronously in an SoC. The advantages and disadvantages of each asynchronous communication schemes are presented. It is determined that using data synchronization approach to interface between two clock domains yield the most robust, design efficient and high performance synchronization interface which can be applied to large scale, complex SoCs. We then discuss several synchronization issues and ways to improve data transfer reliability while maintaining low data transmission latency. In the next chapter, we discuss the details of a mixed-clock shared bus design which provides a highly reliable and low-latency communication interface for subsystems in SoCs.

# Chapter 3

# Mixed-Clock FIFO

This chapter contains two sections: Mixed-clock Interface, and Design of the Mixed-Clock FIFO (MCFIFO).

Section 3.1, Mixed-clock Interface describes the method of interfacing two mixed-timing modules using synchronized handshaking signals. Section 3.2, Design of the Mixed-Clock FIFO, discusses the structure and operation of a MCFIFO.

Many mixed-clock on-chip interconnection schemes have been previously proposed by other research groups. They are designed to provide point-to-point interfaces between mixed-clock modules. In this chapter, we are going to present the design of the MCFIFO, which provides high performance, low latency data transfer between two mixed-timing modules.

## 3.1 Mixed-clock Interface

In the previous chapter, we described a data synchronization system that synchronizes data communication using control signals (*request* and *acknowledge*) that are synchronized to the data transmitter and receiver, respectively. To guarantee that the receiver has time to retrieve, valid data items can be temperately stored in an array of registers. They are held in the registers until the receiver has a chance to validate their existence and reliably read them from the registers. They are removed from the registers only after they have been read by the receiver. In Figure 3.1, a two flip-flop synchronization stage is modified into a FIFO structure with two registers, *dreg0* and *dreg1*. The registers are connected in parallel and each one can read data from the same bus. Control signals govern which register can store the data items from the bus. The *empty* signal informs the receiver if data is available to be read from any of the registers. The circuit controller generates the control signals, and the registers constantly provide feedback of their status to the controller. Based on the feedback, the controller monitors if the registers are full or empty to prevent register overflow or underflow, and adjusts the control signals accordingly.

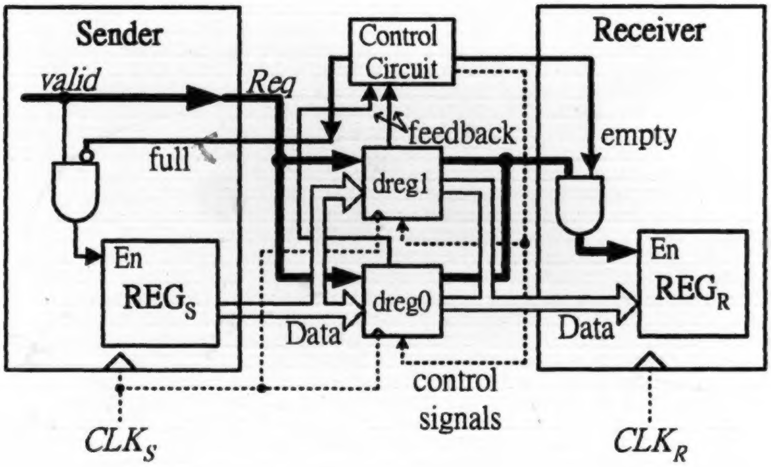


Figure 3.1: Mixed-clock Interconnection with Two Data Registers

In this FIFO structure, the sender does not have to wait for the receiver to acknowledge a data retrieval event. Instead, the data items are clocked into the registers based on the sender's clock, while the controller attempts to identify the status of the registers and to synchronize the control signals. The circuit controller contains a full generator and an empty generator, shown in Figure 3.2. They generate *full* and *empty* signals to inform the sender and receiver of the register status. If the registers are full, *full* is asserted and the sender stops sending data to prevent register overflow. When the registers are empty, the controller sends *empty* = 1 to the receiver to avoid register underflow. This potentially reduces the overall latency of the data transfer. The registers can be viewed as the output buffers of the sender. If the control signals are correctly synchronized, the occurrence of MSS can be reduced.

Chelcea and Nowick [1] noted that the control signals that govern the data transfer need not be synchronized for every data transfer. Synchronization of the control signals is only needed for the receiver when the registers are empty and for the sender when the buffer is full. Consequently, this allows the mixed-clock FIFO to continuously store data and export it almost immediately, while further reducing the transfer latency. The FIFO can be constructed by combining a few predesigned components. It is easy to design since it is technology independent. Its behavior can be modelled using typical HDL languages.

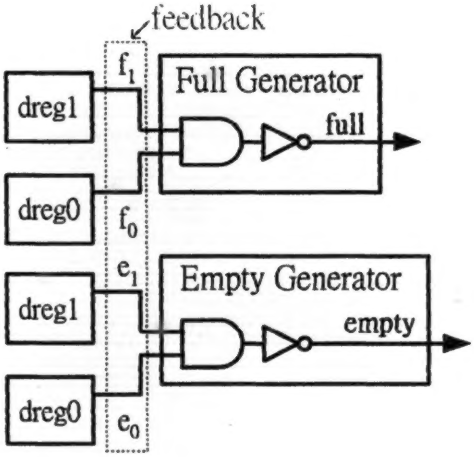


Figure 3.2: The Full and The Empty Generators

## 3.2 Design of the Mixed-Clock FIFO (MCFIFO)

Each MCFIFO can be viewed as a channel, which interfaces a sender and a receiver. The MCFIFO consists of a series of cells that contains data registers and other control logic. The MCFIFO interacts with the sender and receiver through two external interfaces: a put and a get interface. Figure 3.3 explains the overall relationship between the sender, the MCFIFO and the receiver.

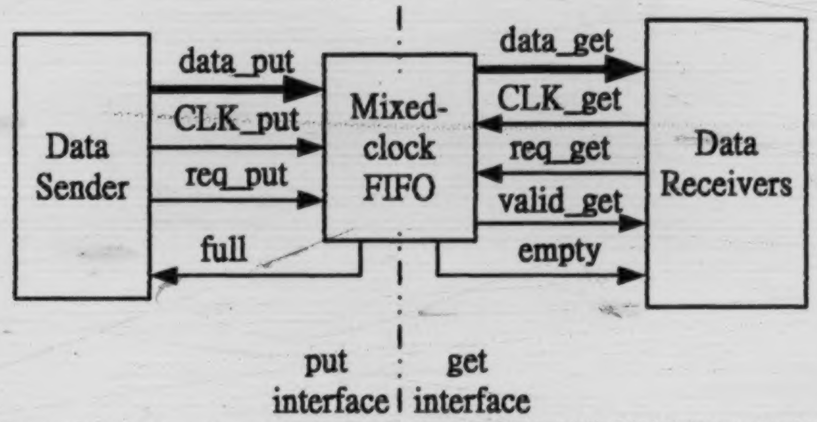


Figure 3.3: Overview of the MCFIFO

Assuming that a data sender operates at the frequency *CLK_put*, the put interface interacts with the sender and is controlled by *CLK_put*. The put interface's inputs consist of a data bus *data_put* and a *req_put* signal for requesting to enqueue a data item, i.e. to put a data item onto the queue formed by the FIFO cells' registers. An output signal *full* is asserted when all of the FIFO cells in the system occupy a data item. Likewise, the get interface is controlled by *CLK_get* that is the operating frequency of the receiver. In addition, the signal *req_get* is used for requesting to dequeue a data item, i.e. to read and remove data from the FIFO cells. To retrieve a data item, it is placed on the *data_get* bus. An output signal *empty* is asserted when all the FIFO cells in the system is empty. Another output signal *valid_get* is always asserted whenever there is a valid data placed onto the *data_get* bus.

Each FIFO is constructed as a circular array of identical cells. An example of a FIFO with four cells is illustrated in Figure 3.4. The same control logic for the data put and get operations is distributed among these cells. This allows concurrency between the two external interfaces. For example, all FIFO cells can detect the arrival of valid data items at the same time by reading *req_put*. Data enqueued onto the circular FIFO architecture is held and is not moved until it is dequeued. Two tokens control the input and output operation of the FIFO. A put token circulating among the FIFO cells enables them to enqueue data items whenever the put token arrives. Similarly, a get token is used to dequeue data items from each FIFO cells. Once a cell has used a token for a data operation, the data item will be removed from the cell and the token is passed to the next cell. Data items are ready to be dequeued as soon as they are enqueued, reducing the latency of the FIFO. These architectures are also highly scalable, as the capacity of the FIFO and the width of the data items can be modified without significant changes to the design.

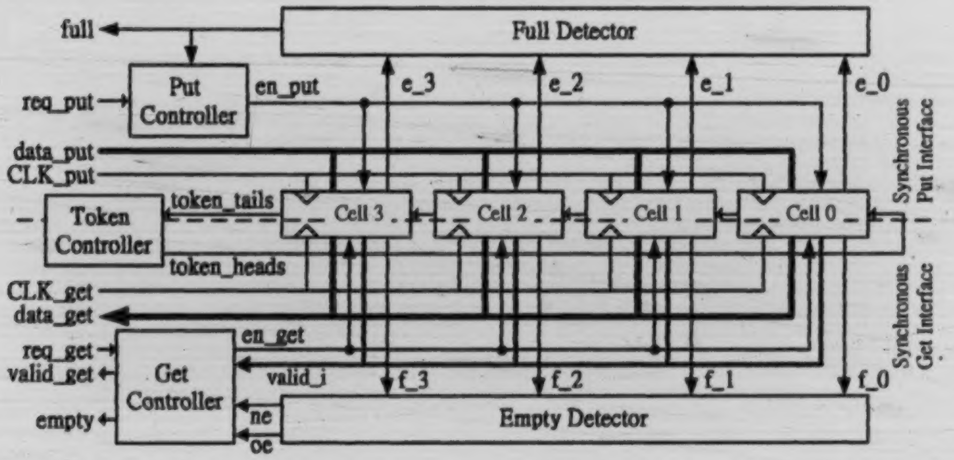

full
Full Detector
req_put
Put Controller
en_put
e_3
e_2
e_1
e_0
data_put
CLK_put
Token Controller
token_tails
token_heads
Cell 3
Cell 2
Cell 1
Cell 0
Synchronous Put Interface
Synchronous Get Interface
CLK_get
data_get
req_get
valid_get
empty
Get Controller
en_get
valid_i
f_3
f_2
f_1
f_0
ne
oe
Empty Detector

Figure 3.4: MCFIFO Architecture [1]

### 3.2.1 FIFO Architecture

In a data put operation, the synchronous put operation starts when a request on *req_put* is received by the put controller on the positive phase of *CLK_put*. A data item is enqueued at the start of the next clock cycle given the FIFO is not full. If the FIFO becomes full, the *full* signal is asserted and any new pending put requests from the sender will be suspended. The FIFO stalls the put token and the enqueuing operation is frozen. Any new data items from the sender must be maintained until *full* becomes 0 or the data items will be lost. A synchronous get operation begins when a request on *req_get* is asserted on the positive edge of *CLK_get* [1]. By the end of that clock cycle, a data item is placed on *data_get* along with *valid_get*. If the FIFO becomes empty, *empty* is asserted and any new pending get requests from the receiver will be suspended until empty becomes 0. After a get request, *valid_get* and *empty* can indicate three outcomes, illustrated in Table 3.1.

Table 3.1: Get Operation Outcomes

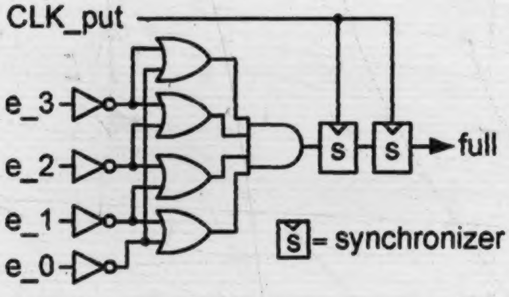
| **Signal Name** | **Value** | **Description** |
|---|---|---|
| *valid_get* | 1 | data item dequeued |
| *empty* | 0 | more items are pending |
| *valid_get* | 1 | data item dequeued |
| *empty* | 1 | FIFO is now empty |
| *valid_get* | 0 | no data item dequeued |
| *empty* | 1 | no data item is available |

Two global control signals, *full* and *empty*, are used to control the data flow of the FIFO and are generated by the full detector and empty detector circuit. Each FIFO cell generates feedback to inform the detectors of their current status. The detector computes the global FIFO status based on the feedback. There are two feedback signals from each cell, one to indicate if the cell is currently filled with a data ($f_i$) and one to indicate if it is empty ($e_i$). The two are updated according to their own time reference. This directly affects how the global control signals are computed. Even though *full* and *empty* are synchronized to their respective interfaces' clocks, they are concurrently read and changed by the put and get interfaces under two different clocks. This condition is similar to that in the two flip-flop synchronizing interface, where *request* and *acknowledge* are controlled by two different clocks. Thus, synchronizers are added to the two detectors.

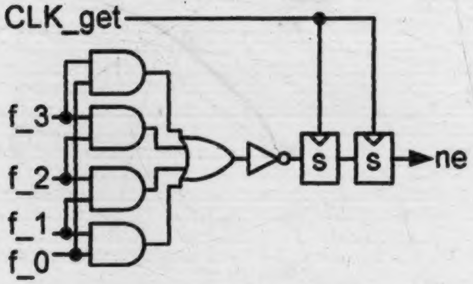
However, the added delay of generating these signals may cause the FIFO to overflow or underflow. We have demonstrated in the last chapter that it is necessary to use at least two synchronizers to minimize metastability of signals. Assuming two synchronizers are added to the detectors, two more clock cycles are then needed to propagate the global control signals to the detector's output ports. This means if the FIFO is full or empty, the external interfaces cannot realize it until two extra clock cycles (of their respective clock) have elapsed. Therefore, a carefully characterized definition of the two control signals has to be determined to avoid overflow, underflow, and deadlock of the MCFIFO.

Let *c* represent the number of synchronizers added to the detectors. For a 4-place MCFIFO, if two synchronizers are added to the full and empty detectors (c=2), the definition of *full* and *empty* must be changed. The FIFO must now consider its state as full when, of the four FIFO cells, either 0 or 1 empty cell is left. It may be considered empty when less than two of the four cells are filled. With the new empty signal, *ne*, the get interface can be controlled to remove data items from queue, issue a new get request, and then wait for response after *c*, or in this case, two *CLK_get*. The new detectors are shown in Figure 3.5a and Figure 3.5b.

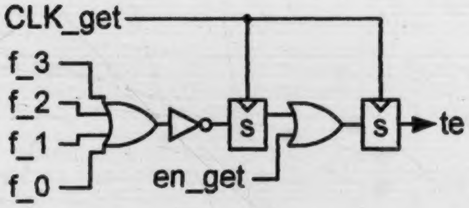
However, with this new *empty* definition, when there is only one data item left in the queue, the early detection of *empty* may cause the FIFO to enter a deadlock state. Thus, an additional detector calls true empty detector, which generates the signal *te*, is introduced to identify this special case. The two empty signals are synchronized with *CLK_get* and are combined through an AND gate to form the global *empty* signal. An implementation of a true empty detector for a 4-place FIFO is shown in Figure 3.5c. The two empty detectors produce two identical signals in all but one


CLK_put
e_3
e_2
e_1
e_0
s
s
full
s = synchronizer

(a) Full Detector


CLK_get
f_3
f_2
f_1
f_0
s
s
ne

(b) Normal Empty Detector


CLK_get
f_3
f_2
f_1
f_0
s
en_get
s
te

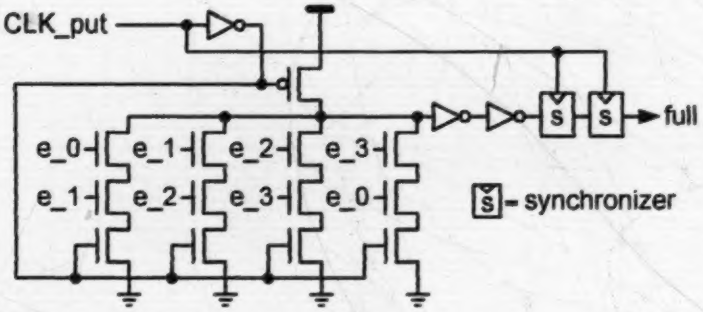
(c) True Empty Detector

Figure 3.5: Global State Detectors for a 4-place MCFIFO

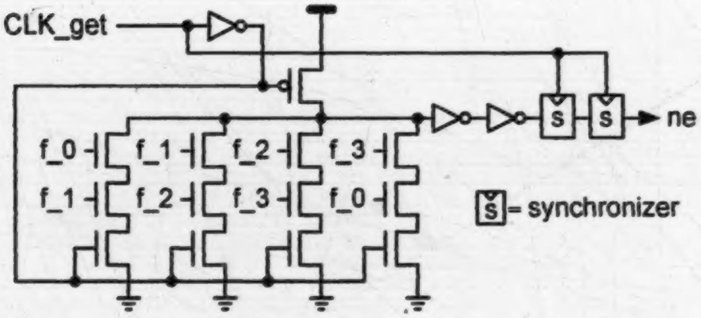
case, when there is exactly one data item in the FIFO. In that case, if another get request is issued by the get interface right after the second-to-last data item has been read, the normal empty detector makes *ne* go low. This allows the get request for the last data item to pass onto the get interface. Subsequently, *ne* goes high which stalls the get interface indefinitely in the next clock cycle, until other data items are enqueued. However, if there is no immediate request for the final data item, *te* forces *empty* to stay 0, preventing the FIFO from stalling until the last data item is dequeued. After the last data has been removed, *ne* can then go high and forces the global *empty* to return back to 1.

According to Equation 2a., two synchronizers are sufficient to provide synchronization for low clock frequencies. However, for higher clock frequencies, more synchronizers and FIFO cells should be added to the FIFO to improve its overall performance. We can use a new technique to include more synchronizers in the three detectors if needed. Suppose that synchronizing the new global control signals ("new *full*", "new *ne*"and "new *te*") requires *c* latches. To prevent overflow, the FIFO will have to be considered full when there are fewer than *c* empty cells in the FIFO. Likewise, to prevent underflow, the FIFO is considered empty when there are fewer than *c* full cells in the FIFO.

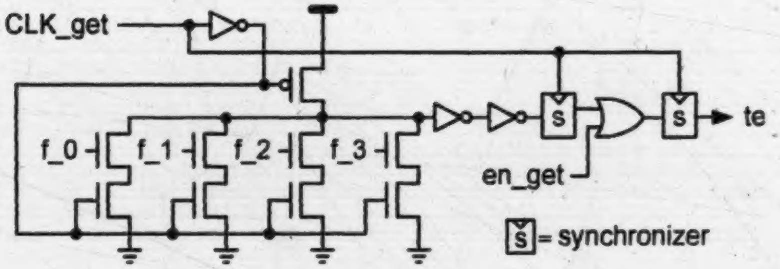
Although synchronizing the two global control signals to their respective clock signals provides a good MTBF rate, it does not eliminate the synchronization failure completely. However, by further optimizing the detectors' circuits, we can guarantee that even if the global control signals are incorrectly read due to metastability, the MCFIFO does not perform an illegal operation. To achieve that, the static logic gates used to implement the detectors are replaced by dynamic logic [1], as shown in Figure 3.6. Figure 3.7 illustrates the global state detectors for an eight-place FIFO. With the new implementation, the global states are pre-charged to be high


CLK_put
e_0
e_1
e_2
e_3
e_1
e_2
e_3
e_0
full
S
= synchronizer

(a) Full Detector


CLK_get
f_0
f_1
f_2
f_3
f_1
f_2
f_3
f_0
ne
S
= synchronizer

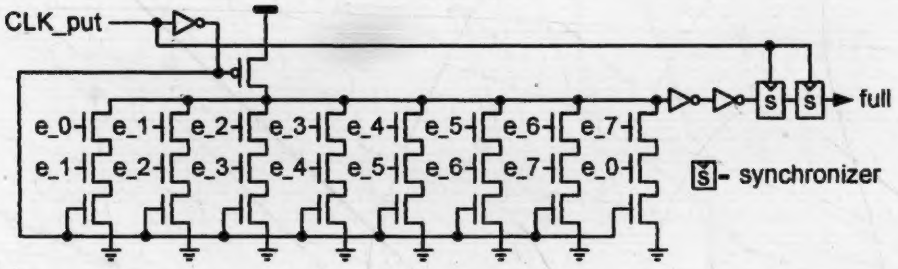
(b) Normal Empty Detector


CLK_get
f_0
f_1
f_2
f_3
te
en_get
S
= synchronizer

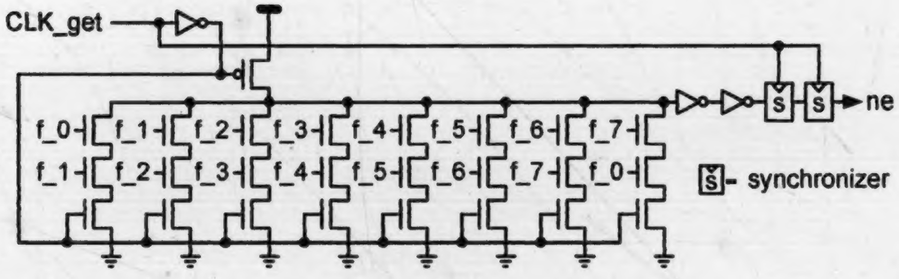
(c) True Empty Detector

**Figure 3.6: Global State Detectors Implemented with Dynamic Logic for a 4-place MCFIFO [1]**

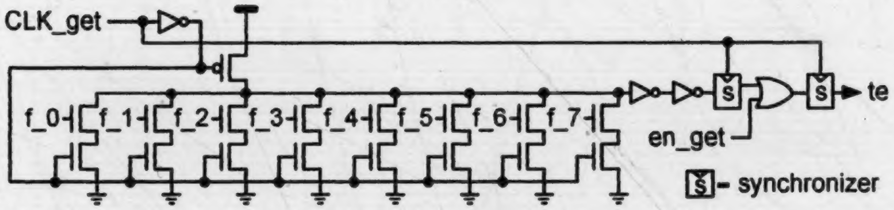
during the positive phase of every clock cycle. Then, during the negative phase of the same clock cycle, the detectors' logic computes and updates the global state of


CLK_put
e_0
e_1
e_2
e_3
e_4
e_5
e_6
e_7
e_1
e_2
e_3
e_4
e_5
e_6
e_7
e_0
S
S
full
S - synchronizer

(a) Full Detector


CLK_get
f_0
f_1
f_2
f_3
f_4
f_5
f_6
f_7
f_1
f_2
f_3
f_4
f_5
f_6
f_7
f_0
S
S
ne
S - synchronizer

(b) Normal Empty Detector

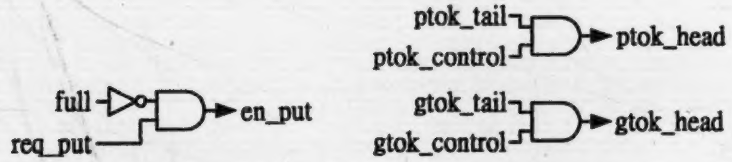

CLK_get
f_0
f_1
f_2
f_3
f_4
f_5
f_6
f_7
S
S
te
en_get
S - synchronizer

(c) True Empty Detector

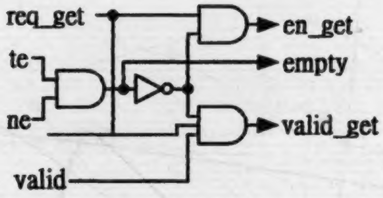
Figure 3.7: Global State Detectors Implemented with Dynamic Logic for a 8-place MCFIFO

the FIFO based on the feedback from each cell. This guarantees that the global *full* and *empty* signals can only enter MSS on a transition from *full/empty* to *not-full/not-empty*. Therefore, an incorrect read of the control signals results only in stalling the interface for one extra clock cycle, but prevents FIFO overflow/underflow or loss of data. This increases the latency of the data transfer but significantly reduces synchronization failure to occur.

In addition to the external interfaces and global state detectors in each FIFO, three external controllers are also designed as parts of the synchronous interface's components. The put and get controllers process data operation requests based on the global state of the FIFO and pass those requests onto their respective interfaces. For example, the put controller, in Figure 3.8a, passes requests to the FIFO but withholds those requests when the FIFO is full. Similarly, the get controller, in Figure 3.8c, forwards all the get requests unless the FIFO is empty. The put and get controllers enable and disable the put and get operation and the movement of the put and get tokens, respectively, in the FIFO. The token controller is used to initialize and generate the put and get tokens upon system reset.


full
req_put
en_put
ptok_tail
ptok_control
ptok_head
gtok_tail
gtok_control
gtok_head

(a) put controller (b) token controller


req_get
te
ne
valid
en_get
empty
valid_get

(c) get controller

Figure 3.8: MCFIFO Controllers [2]

### 3.2.2 FIFO Cell's Implementation

A detailed implementation of a cell is shown in Figure 3.9. The cell begins in an empty state initially and without any tokens. Upon system reset, the token controller generates two token heads: a put token head and a get token head. The right most cell (Cell 0 of Figure 3.4) waits to receive the put token head from the token controller on the positive edge of *CLK_put* and waits for the sender to place a valid data item on the *data_put* bus. A valid data item is indicated to all cells by *en_put*=1. The cell enables the data register to latch the data item and *en_put* as the data validity bit. At the same time, it indicates that the cell is full with $f_i = 1$ and enables the $EnTokDff_{po}$ to latch the put token. Then on the next positive edge of *CLK_put*, the data item and the validity bit are finally stored in the data register and the put token is passed to the cell on the left (Cell 1). The data item and valid bit are now available to be read by the receiver. The behavior for dequeuing data is similar.

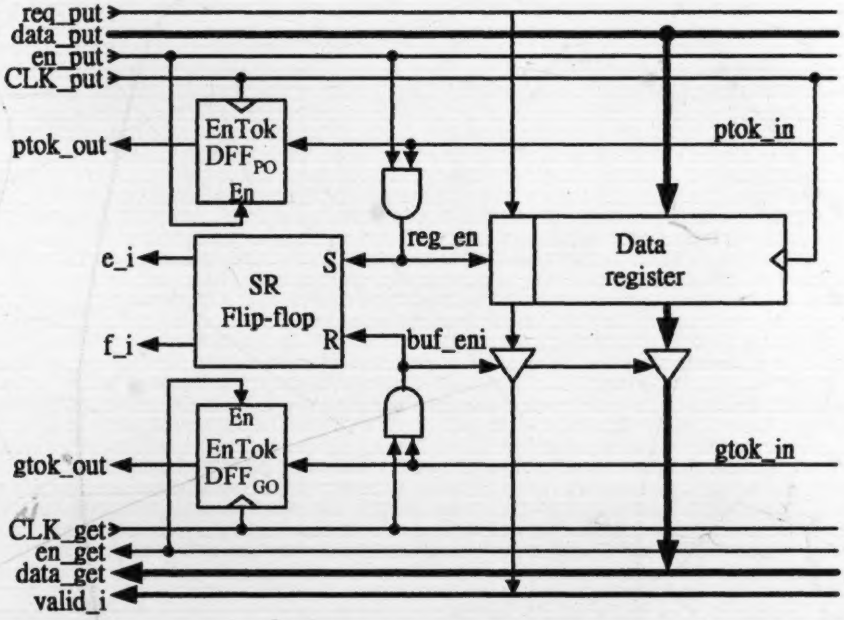

req_put
data_put
en_put
CLK_put
ptok_out
EnTok DFF PO
En
ptok_in
reg_en
Data register
e_i
S
SR Flip-flop
R
f_i
buf_eni
En
EnTok DFF GO
gtok_out
gtok_in
CLK_get
en_get
data_get
valid_i

Figure 3.9: FIFO Cell's Implementation [2]

The cell waits to receive the get token and waits for the receiver to request a data item. When both signals are presented, the cell enables placing of the data item and valid bit on the *data_get* and *valid_i* tri-state bus, respectively. At the same time, the cell forces $e_i$ to go high and enables $ETDFF_{go}$ to latch in the get token. On the next positive edge of *CLK_get*, the get token is passed to the left cell (Cell 1). This process carries the tokens from the first cell to the last cell (for example, Cell 3 of Figure 3.4) of the FIFO and then back to the first one. It repeats continually until the data transfer is completed.

## 3.3 Summary

In this chapter, we introduce the structure and function of the MCFIFO. The MCFIFO design is briefly described in the early sections. We describe the design of several controllers that governs the operation of the MCFIFO. We then discuss how to modify the definitions of the *empty* and the *full* signals so that the MCFIFO has the ability to prevent FIFO overflow and underflow. An additional empty detector is introduced to the MCFIFO design for resolving deadlock situation of the FIFO, which occurs when only one data item is queued among all FIFO cells. Finally, a throughout description is provided to explain the implementation and the operation of the MCFIFO cell.

# Chapter 4

# Shared Bus Design Using MCFIFO

This chapter contains three sections: Broadcasting of Data Items Using the MCFIFO, Data Broadcasting Shared Bus (DBSB), and Mixed-Clock Shared Bus (MCSB).

Section 4.1, Broadcasting of Data Items Using the MCFIFO, describes the architecture of the data broadcasting FIFO (DBFIFO), how it functions, and its applications. Section 4.2, Data Broadcasting Shared Bus (DBSB), discusses how the DBFIFO concept can be applied to a data broadcasting shared bus design. Section 4.3, Mixed-Clock Shared Bus (MCSB), introduces a shared bus design that utilizes a MCFIFO to provide a multi-point interconnection in an SoC environment.

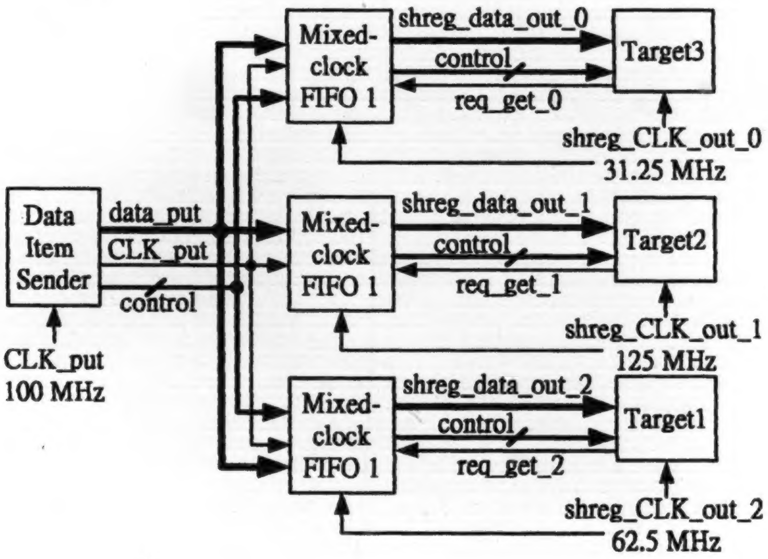
## 4.1 Broadcasting of Data Items Using the MCFIFO

The MCFIFO described in the previous chapter accommodates point-to-point interconnection between two mixed-timing modules. However, to use such a design in an SoC environment, multiple MCFIFOs must be employed to provide interconnection between multiple modules, which adds large area overhead to the system. We will discuss how the MCFIFO can be modified to support multiple point interconnections, with reduced area overhead, while maintaining the design's low latency or high robustness characteristics. First, we focus on using the MCFIFO to provide the data broadcasting function. The FIFO can be enhanced to handle broadcasting of data items from a single source to multiple targets operating in different rational clock domains. Figure 4.1b shows an example of the improved FIFO system that allows data items to be broadcasted to three targets running at different clock domains. If a point-to-point topology were employed to connect the three receivers to the sender using only MCFIFOs, one would need a system as shown in Figure 4.1a, where three FIFOs are implemented and form a mesh network. If more modules are combined into the system, even more MCFIFOs are needed. That could add an unnecessarily large amount of area overhead to the overall system.

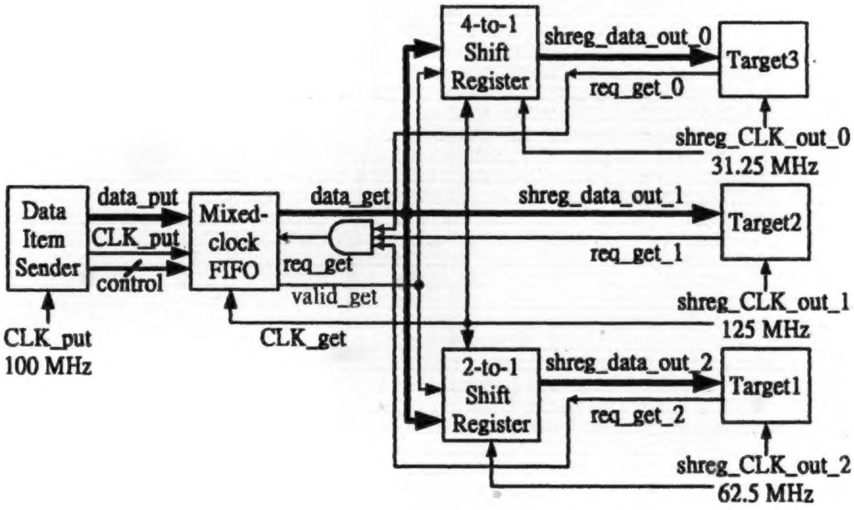
Assuming each local clock, *shreg_CLK_out_i*, runs at pre-determined rational multiples of *CLK_get*, shift registers can be used to interface between the receivers and the MCFIFO. Since the shift registers are generally much smaller designs than the MCFIFO, logic and area overhead can be reduced. We called this new FIFO architecture a *data broadcasting FIFO* (DBFIFO).

The DBFIFO contains two stages: a FIFO stage consists of an MCFIFO, and a data broadcasting stage consists of two or more shift registers of different lengths, each interfacing with a broadcasting target. Each of the FIFO stage and the data broadcasting stage contains a put and get interface, as illustrated in Figure 4.2. The put interface of the data broadcasting stage is actually the extension of the get interface of the FIFO stage. It is controlled by *CLK_get*. The broadcasting targets request data items from the MCFIFO with *req_get_i*. The shift registers then receive *data_get* and *valid_get* from the MCFIFO.

Since the broadcasting targets operate at different clock rates, data is bundled together before it is delivered to the targets. For example, target 1 in Figure 4.1b, is operating at *shreg_CLK_out_2*, or half the clock rate of *CLK_get*. At that rate, in one *shreg_CLK_out_2* clock cycle, two data items could have been read from *data_get*. Therefore, a serial-in, parallel-out shift register with a 2-to-1 ratio is used to clock in


Data Item Sender
data_put
CLK_put
control
CLK_put
100 MHz
Mixed-clock FIFO 1
Mixed-clock FIFO 1
Mixed-clock FIFO 1
shreg_data_out_0
control
req_get_0
Target3
shreg_CLK_out_0
31.25 MHz
shreg_data_out_1
control
req_get_1
Target2
shreg_CLK_out_1
125 MHz
shreg_data_out_2
control
req_get_2
Target1
shreg_CLK_out_2
62.5 MHz

(a) Multiple Mixed-Clock FIFO (M-MCFIFO)


Data Item Sender
data_put
CLK_put
control
CLK_put
100 MHz
Mixed-clock FIFO
data_get
req_get
valid_get
CLK_get
4-to-1 Shift Register
shreg_data_out_0
req_get_0
Target3
shreg_CLK_out_0
31.25 MHz
shreg_data_out_1
req_get_1
Target2
shreg_CLK_out_1
125 MHz
2-to-1 Shift Register
shreg_data_out_2
req_get_2
Target1
shreg_CLK_out_2
62.5 MHz

(b) Data Broadcasting FIFO

Figure 4.1: Data Broadcasting Architecture

two data items at *CLK_get*, bundle the two data items into one, and then output it to target 1 at the rising edge of *shreg_CLK_out_2*. Table 4.1 illustrates the relationships

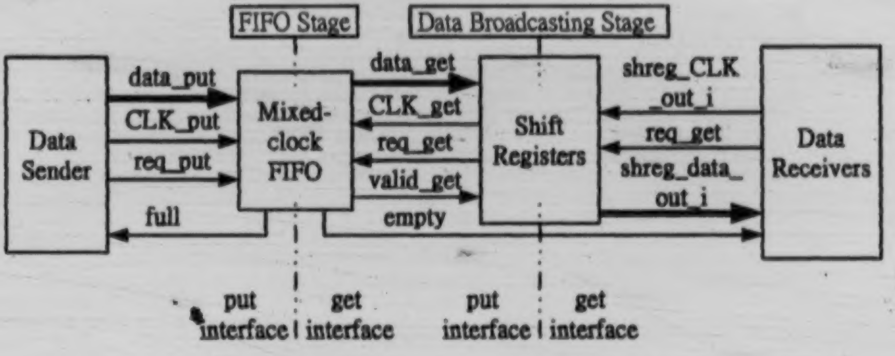

FIFO Stage
Data Broadcasting Stage
Data Sender
data_put
CLK_put
req_put
full
Mixed-clock FIFO
data_get
CLK_get
req_get
valid_get
empty
Shift Registers
shreg_CLK _out_i
req_get
shreg_data_ out_i
Data Receivers
put interface
get interface
put interface
get interface

Figure 4.2: Synchronous Interfaces of the DBFIFO

between the three clock domains with respect to *CLK_get* for the example shown in Figure 4.1b.

Table 4.1: Timing Relationships between The Three Subsystems

| Clock Signal | Clock Speed | Clock Ratio |
|---|---|---|
| *CLK_get* | 125MHz | 1 |
| *shreg_CLK_out_0* | 31.25MHz | 4/1 |
| *shreg_CLK_out_1* | 62.5MHz | 2/1 |
| *shreg_CLK_out_2* | 125MHz | 1 |

### 4.1.1 Shift Register

We first consider *shreg_CLK_out_i* to have the same or slower clock rate than *CLK_get*. This is the case illustrated in Figure 4.1b. If the clock edges of *shreg_CLK_out_i* and *CLK_get* are in phase with each other, a serial-in, parallel-out shift register (SPSR) that can interface a 4-to-1 clock ratio is shown in Figure 4.3. This clock ratio implies *CLK_get* is four times faster than *shreg_CLK_out_i*. The shift register consists of (4/1 clock ratio) four shift register cells.

A SPSR is denoted as a type-A shift register. The 4-to-1 type-A shift register takes in four inputs: *data_get*, *valid_get*, *CLK_get* and *shreg_CLK_out*. An M-bit data item enters the shift register through the *data_get* bus. A shift register that can interface an N-to-1 clock ratio will contain N shift register cells. Consequently, an NxM-bit data item will exit the shift register through the *shreg_data_out* bus every shreg_CLK_out_i. For cases where *shreg_CLK_out_i* are rational multiples

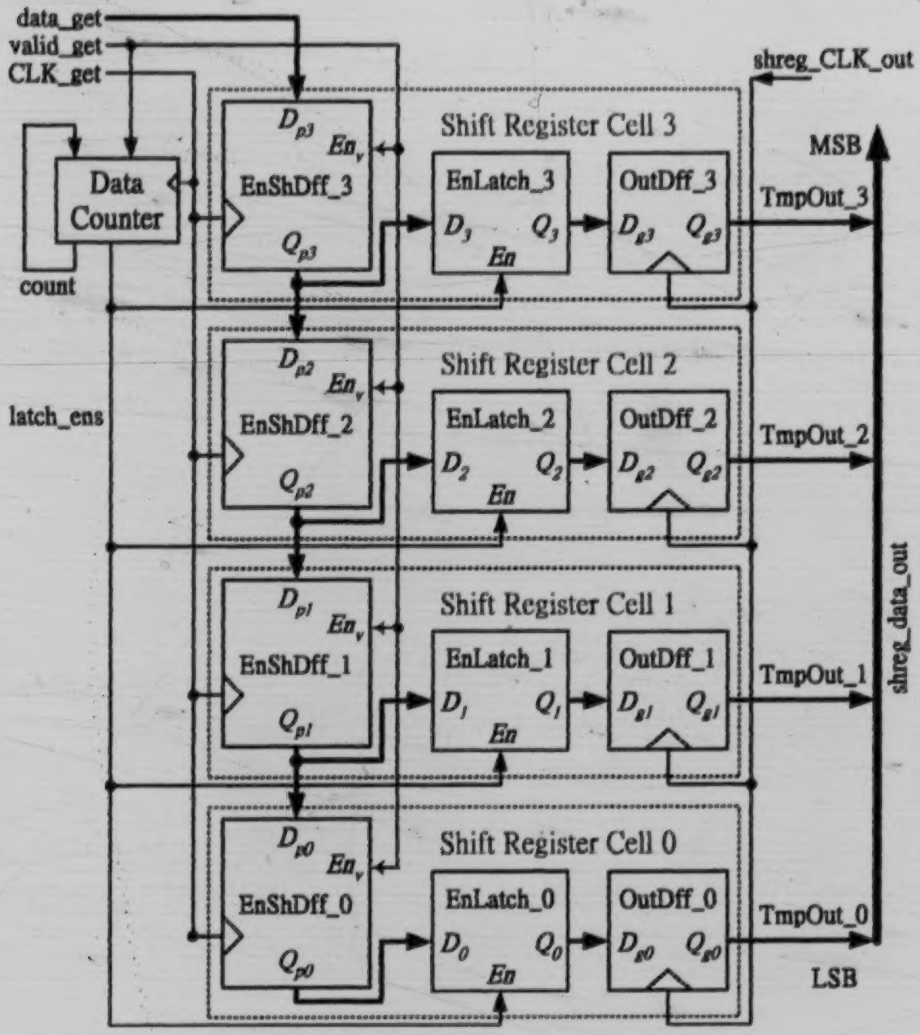

data_get
valid_get
CLK_get
shreg_CLK_out
Data Counter
count
latch_ens
Shift Register Cell 3
Shift Register Cell 2
Shift Register Cell 1
Shift Register Cell 0
EnShDff_3
EnShDff_2
EnShDff_1
EnShDff_0
EnLatch_3
EnLatch_2
EnLatch_1
EnLatch_0
OutDff_3
OutDff_2
OutDff_1
OutDff_0
TmpOut_3
TmpOut_2
TmpOut_1
TmpOut_0
MSB
LSB
shreg_data_out

Figure 4.3: Type-A 4-to-1 Shift Register

of, but faster than, *CLK_get*, a shift register may not be needed, since from the target's perspective, it implies a broadcasted data item is made available for more than one receiver's clock cycle. However, to prevent the receiver from reading a data item more than once, a parallel-in, serial-out shift register (PSSR) can be used to interface between the MCFIFO and the broadcasting targets. A PSSR is denoted as a type-B shift register.

A type-B shift register that interfaces a 1/4-to-1 clock ratio, or 1-to-4 clock ratio is shown in Figure 4.4. The clock ratio implies *shreg_CLK_out_i* is running four times faster than *CLK_get*. Therefore, instead of grouping data together as in a type-A shift register, the type-B shift register breaks down an input data items into 4 equal segments. It then shifts out each segment at the rising edge of *shreg_CLK_out_i*.

### 4.1.2 DBFIFO Protocol

The FIFO stage functions in the exact same fashion as described in Section 3.2. In this section, we describe the operation of the data broadcasting stage in details.

In the beginning of a data broadcast, the sender issues a broadcast request to all targets. The targets respond by asserting their *req_get_i* high. The three *req_get_i* signals are *AND*ed together and form *req_get*, which is sent to the MCFIFO. The sender passes the data onto the FIFO stage through *data_put*. The MCFIFO transfers the data to its output, and informs the shift registers in the data broadcasting stage of the availability of data with *valid_get*. Refereing to Figure 4.3, a N-to-1 shift register contains N shift register cells. Each cell consists of three components: an *EnShDff*, an *EnLatch* and one *OutDff*. An *EnShDff* is a d flip-flop with an extra enable control signal, $En_v$. $En_v$ is connected to *valid_get* from the MCFIFO. Whenever a valid data item is placed on *data_get*, the validity bit enables *EnShDff* to latch the data at the positive edge of *CLK_get*. Then on the negative phase of *CLK_get*, the data is stored in*EnShDff*. That data item is shifted to the next *EnShDff* every time a new valid data item arrives.

The arrival of the validity bit triggers the data counter to increment its count by

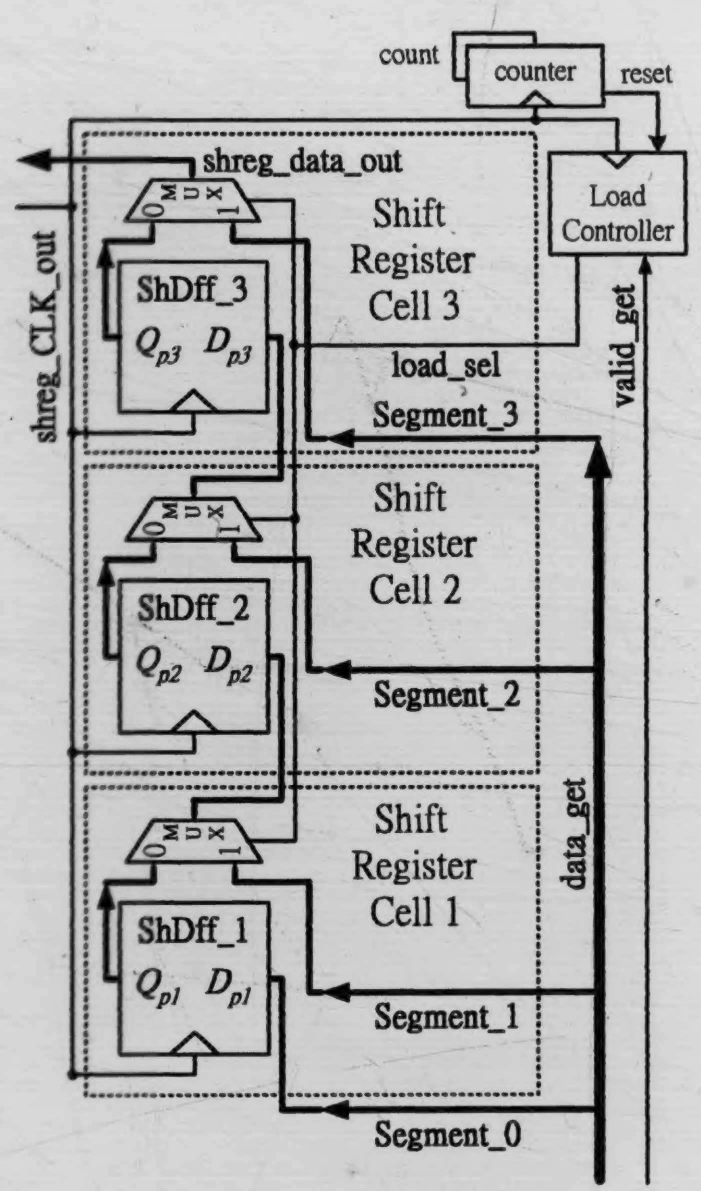

count
counter
reset
Load Controller
shreg_data_out
shreg_CLK_out
MUX
0
1
ShDff_3
$Q_{p3}$ $D_{p3}$
Shift Register Cell 3
load_sel
Segment_3
valid_get
ShDff_2
$Q_{p2}$ $D_{p2}$
Shift Register Cell 2
Segment_2
data_get
ShDff_1
$Q_{p1}$ $D_{p1}$
Shift Register Cell 1
Segment_1
Segment_0

Figure 4.4: Type-B 1-to-4 Shift Register

1. When *count* equals *clock_ratio*, *latch_ens* is asserted. For a N-to-1 type-A shift register, N data items would have been collected in N *EnShDff*s when *latch_ens* is asserted. The assertion of *latch_ens* causes all of the N data items to be loaded into

the respective EnLatch_i of the shift register cells. After that, they wait for the positive edge of *shreg_CLK_out_i* to arrive and exit the shift register through *OutDff*s and onto a NxM-bits data bus, *shreg_data_out*, on the next cycle. The *EnLatch*s are used to store data until the arrival of the positive edge of *shreg_CLK_out_i*. Although the input and output clocks are rational multiple of each other, their clock edges may not arrive at the same moment. The *EnLatch*s allows the two edges to be separated by no larger than N CLK_get cycles. However, the positive edge of *shreg_CLK_out_i* is assumed to be in phase with *CLK_get*.

It is important to noted that since the rising edges of *CLK_get* are assumed to be in phase with the rising edges of *shreg_CLK_out_i*, the type-A shift register design can be further simplified by removing the *EnLatch*s and/or *OutDff*s. If the clock edges are not in phase, the shift register may enter the MSS resulting in synchronization failure.

For a 1-to-N type-B shift register (see Figure 4.4), a data item is broken down into N segments. The arrival of *valid_get* indicates a valid data item is available on *data_get* and triggers the load controller to assert *load_sel* to high for one *shreg_CLK_out_i* cycle. In a 1-to-N type-B shift register, an input data is divided into N segments. For example, a 1-to-4 shift register divides a data into 4 segments as shown in Figure 4.4. During the one clock period when *load_sel* is high, *Segment 3* is placed directly onto *shreg_data_out*. The remaining segments are latched into the *ShDffs*. On the next *shreg_CLK_out*, *load_sel* returns to low. The multiplexers in each shift register cell select the inputs from the *ShDff*s as their outputs. For the next three clock cycles, the shift register shifts out the remaining segments. The counter increments its count on each *shreg_CLK_out*. Every four cycles, the counter produces a signal to reset the controller. After the reset, the controller is ready to assert *load_sel* high again. If a new valid data item arrives, the same operation is repeated. If no new valid data appears, *valid_get* stays low and the target stops reading from *shreg_data_out* until the next valid data arrives.

### 4.1.3 DBFIFO Applications

In [36, 37], a core-based test generation approach for random logic core circuits that utilizes the concept of test pattern broadcasting is proposed. In this approach, a traditional scan chain design is modified into a new test architecture so test vectors can be broadcasted to all modules to be independently tested. The overall scan depth and test time are, therefore, significantly reduced in this new architecture. The DBFIFO allows broadcasting of normal data items or core-based test patterns to the cores of an SoC design for parallel computation or parallel scan test. We also use the DBFIFO as a first step for other work presented in this chapter, such as the design of shared bus architectures for multi-point, mixed-timing interface in SoCs.

### 4.1.4 Shared Bus

Inter-module interconnection within SoCs is becoming more and more critical as the complexity of SoCs grows and their feature size decreases. A shared system bus is a key feature of modern SoC design methodologies. The on-chip buses must be designed to be sufficiently flexible and robust in order to fulfill the wide variety of performance specification of SoCs. Later in this chapter, we introduce two shared bus architectures that allow efficient and flexible multi-point interconnection between mixed-timing modules in SoCs, using MCFIFOs.

Traditionally, the interfaces to a shared bus may be centralized or distributed. The centralized approach groups all the bus interfaces and control components in a central hub forming a star network. Each device has dedicated point-to-point connection to the hub. The distributed approach represents the more conventional view of a shared bus. This approach places the bus interface near the devices so as to minimize the size of the point-to-point links between the device and its interface. It leads to a higher loading on the bus and slower operation, but typically gives a much smaller implementation due to the reduction in wiring.

The distributed arbitration techniques found in off-chip buses or networks such as SCSI [38] and Ethernet are not suitable for low-power SoCs because they permit drive-clashes or polling of the arbitration signals [3]. Synchronous on-chip buses, like the Advanced Microprocessor Bus Architecture (AMBA) [4], use a centralized arbitration system with request and grant handshaking signals, connecting modules to the central arbiter (arbitrator). This approach is more expensive in area but minimizes the length of the shared lines, and hence their load, allowing faster edges and weaker drivers. Our shared bus utilizes a centralized approach since the mixed-clock interface is performed by a central MCFIFO(s), and the connections to the MCFIFO are realized using a multiplexed control and data-path drive network, which is governed by a centralized bus controller.

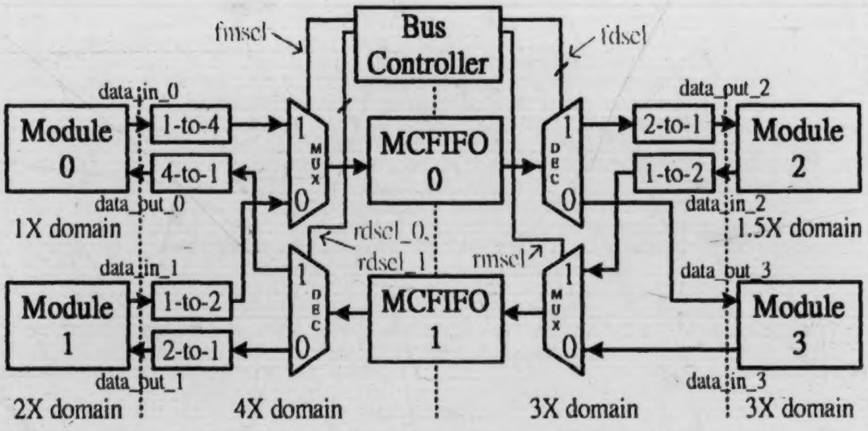
Multiple point data communications on a shared bus involve bi-directional data transfers. There are modules that initiate communication and targets that obtain the communication request and react accordingly. When a channel has more than one initiator drive clashes, data corruption and signaling failure may occur. It is imperative that only one initiator acts upon the channel at any one time. On the other hand, when a channel is connected to more than one target, extra functionality is required to determine which target should receive and respond to each cycle, and to ensure that drive and signaling clashes between targets do not occur. Extra control circuitry is therefore required to determine which module should use the channel and to ensure correct operation of the shared bus. This task is accomplished using a bus controller which acts between modules to determine which owns the channel. In the following two sections, details of shared bus architectures constructed using

MCFIFOs are presented.

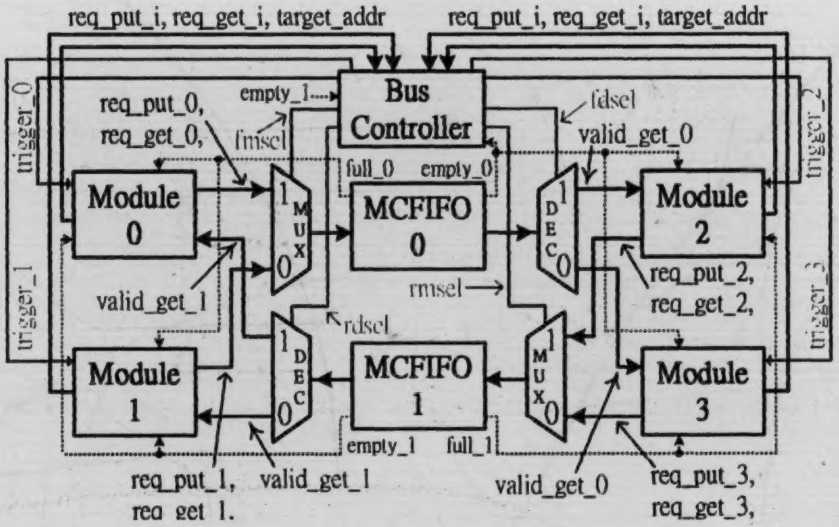
## 4.2 Data Broadcasting Shared Bus (DBSB)

The DBFIFO can be further enhanced to form a *data broadcasting shared bus* (DBSB) using a series of SPSR and PSSR. Figure 4.5 presents an example of a simple SoC design with a DBSB connecting four modules. Here, we have two groups of modules operating at different rational clock frequencies. Module 0 and 1 operate at clock frequencies that are pre-determined rational multiples of regional clock 1 (4x). Module 2 and 3 operate at rational clock frequencies of regional clock 2 (3x). The frequency relationship between different clocks is indicated in Figure 4.5a. If 1x represents a clock frequency of 100MHz, then 2X implies two times the speed of 1x, or 200MHz. Similarly, 1.5x in Figure 4.5a implies Module 2 has a clock rate 1.5 times the speed of 1x, which is 150MHz.

The shared bus is structured such that Module 0 and Module 1 are interconnected with Module 2 and Module 3 bidirectionally through MCFIFO_0, and MCFIFO_1. Module 0 and 1 can either send or receive data from Module 2 and 3 or broadcast data to Module 2 and 3 simultaneously. Similarly, Module 2 and 3 can either send or receive data from Module 0 and 1 or broadcast data to Module 0 and 1 at the same time. The *data_put* and *data_get* of MCFIFO are assumed to have fixed width. Because the modules are operating at rational clock frequencies, the same shift register approach used in DBFIFO can be applied to DBSB. Interconnection between Module 0 and Module 1, and Module 2 and Module 3 are provided using type-A and type-B shift registers only. As in the case of DBFIFO, the correct operation of the DBSB requires the multiple rational clock signals to have clock edges that align with the edges of their clock source. For example, if clock signals of Module 0 and Module 1 were derived from the same clock source, they are


Bus Controller
fmsel
fdsel
data_in_0
1-to-4
Module 0
4-to-1
data_put_0
1X domain
MUX
MCFIFO 0
DEC
data_put_2
2-to-1
Module 2
1-to-2
data_in_2
1.5X domain
rdsel_0,
rdsel_1
rmsel
data_in_1
1-to-2
Module 1
2-to-1
data_put_1
2X domain
DEC
MCFIFO 1
MUX
data_put_3
Module 3
data_in_3
4X domain
3X domain
3X domain

**(a) Data Path of DBSB**


req_put_i, req_get_i, target_addr
req_put_i, req_get_i, target_addr
trigger_0
trigger_1
trigger_2
trigger_3
req_put_0,
req_get_0,
empty_1
fmsel
Bus Controller
fdsel
valid_get_0
full_0
empty_0
Module 0
MUX
MCFIFO 0
DEC
Module 2
valid_get_1
rmsel
rdsel
req_put_2,
req_get_2,
Module 1
DEC
MCFIFO 1
MUX
Module 3
empty_1
full_1
req_put_1,
req_get_1,
valid_get_1
valid_get_0
req_put_3,
req_get_3,

**(b) Control Path of DBSB**

Figure 4.5: Data Broadcasting Shared Bus

assumed to have clock edges that are in phase with each other.

### 4.2.1 Bus Controller

To govern which module has access priority to the bus, a controller is constructed. Instead of building a complicated arbiter circuit to determine which modules first requests bus access, a simpler control circuit is used to grant bus access permission to modules. Each module is assumed to have a fixed priority to use the bus. The assignment of their priority is arbitrary. Module 0 in Figure 4.5 is assigned to have the highest access priority and Module 3 has the lowest. By doing so, the controller design is greatly simplified, and we can focus our research effort on integrating the MCFIFO into the mixed-clock shared bus architectures. The bus controller in Figure 4.6 takes in several inputs from each module, *req_put_i* and *target_addr*. If a module wants to deliver data to a target via the MCSB, it asserts *req_put* and it sends the bus controller the address of the target, *target_addr*. A central decoder in the bus controller constantly waits for a valid address and then signals the addressed target to respond, using a separate target-select signal, *trigger_i*. Referring to Figure 4.5b, two sets of control signals to the bus traffic control multiplexer (MUX) and decoder (DEC) (*fsel*, *fdsel*, and *rsel*, *rdsel*) are generated by the bus controller to properly direct the data transfer from the initiator to the target(s). *fsel*, *fdsel* are used to control the MUX and DEC that are associated with MCFIFO_0, and *rsel*,

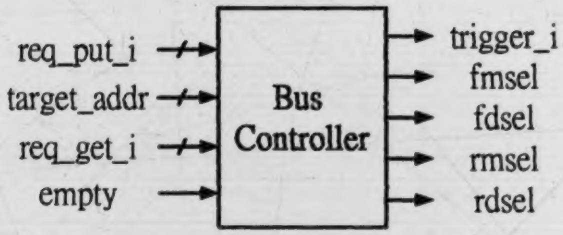


Figure 4.6: Bus Controller

*rdsel* are used to control the MUX and DEC associated with MCFIFO_1. In this example, each traffic control signal is one-bit in width. However, if more modules are presented in the system, the width of control signals must be increased.

### 4.2.2 Shift Register

The type-A and type-B shift registers are used to provide serial-in, parallel-out or parallel-in, serial-out data conversion in the fashion described in Section 4.1.1. When they are used to provide interface between the module and the put interface of the MCFIFO, they are called input shift registers, as opposed to the output shift registers which are connected to the get interface of the MCFIFO. Like the output shift register, an input shift register can be either type-A or type-B shift register, depending on the clock ratio of the data sender and the put interface of the MCFIFO. However, unlike the output shift register, it takes *req_put* instead of *valid_get* as one of the input signals.

### 4.2.3 Bus Traffic Control

To avoid drive clashes during the handover of the bus between initiators, the data path drive is governed by traffic control MUXs on each transfer direction. It is understood that some bus architectures use a tri-state data-path to allow for a smaller implementation. However, a multiplexed data-path control is more suitable for SoC synthesis [3] and is therefore employed in the DBSB design. Similarly, traffic control DECs are used to govern which targets have the right to read from the MCFIFOs. To allow data broadcasting, a DEC may permit data to be placed on all of its outputs. The control signals of the MUXs and DECs are generated by the bus controller.

### 4.2.4 DBSB Protocol

The DBSB begins in an empty state initially. The bus controller waits for an initiator to make a request for the bus. It checks any module requests for the bus in the order of their priority, such as, from Module 0 to Module 3 in our example. A module prepares a data transfer by producing valid data before arresting *req_put_i*. The initiator then requests to put data on the bus with a valid address. The bus controller receives the *req_put_i* and the address of the intended receiver from the initiator. It also computes and asserts the corresponding *trigger_i* and traffic control signals. The type-A or type-B shift register processes the data according to the clock ratio between the MCFIFO put interface's and transmitter's clock. The put interface of the MCFIFO_i then receives *req_put_i* and the data from *data_put* via the traffic control MUXs.

When the target module receives the trigger, it responds by asserting its *req_get_i*. The MCFIFOs operate with their own *CLK_put* and *CLK_get* clocks. They enqueue and dequeue the data as described in Section 3.2.1. After that, the data item along with the validity bit are sent to the target via the traffic control DECs. The output shift register manipulates the output data rate and data width according to the clock ratio of the target and the get interface of MCFIFO_i. Finally, the targeted receiver obtains the data and responds accordingly. The initiator indicates the end of bus access by setting *req_put* to low.

Once the MCFIFO is empty and the target set its *req_get* signal to low, the bus controller assigns bus access to the next initiator. If the target is required to reply to the initiator with data, it requests for bus access. After that, it follows the same procedures described above to transfer the data across. The MCFIFOs broadcast *full_i* and *empty_i* signals to all the modules simultaneously. The MCFIFOs stall any put or get operation if their respective *full_i* or *empty_i* are asserted. The initiator has to maintain the data items to be sent if MCFIFO_i is full or the data is lost.

## 4.3 Mixed-Clock Shared Bus (MCSB)

One limitation of the DBFIFO and the DBSB is that it works for modules that are operating at rational multiples of some clock frequencies. It also requires shift registers for proper operation. If we omit the broadcasting capability, we can remove the shift registers and design a *Mixed-Clock Shared Bus* (MCSB). A MCSB permits multi-point interconnection of modules running at multiple clock rates, thus it is more flexible than the DBSB.

Figure 4.7 presents an architecture of a MCSB. It maintains the use of the centralized interface and continues to employ a multiplexed data-path drive approach, but abandons the use of the shift registers. Instead, the initiators put the request signals and the data directly to the MCFIFO via the traffic control MUX. The transmitters and receivers are responsible for providing the *CLK_put* and *CLK_get* of the MCFIFO_i, respectively. An extra pair of MUXs is used to select clock signals from the transmitters and the receivers. A bus controller similar to the one of the DBSB can also be used for the MCSB.

### 4.3.1 MCSB Protocol

An initiator begins a data transfer by sending *req_put* and *target_addr* to the bus controller. The controller detects the request and generates the corresponding control signals: *fsel*, *fdsel*, and *rdsel*. It is noted that *rsel* is not used in MCSB. The control signals trigger the traffic control MUXs to connect the corresponding *req_put*, *data_put* and *CLK_put* of the initiator to the MCFIFO. Similarly, the control signals force the traffic control DECs to connect the corresponding *req_get*, *data_get* and *data_get* of the target to the MCFIFO. The end of each bus access is indicated by

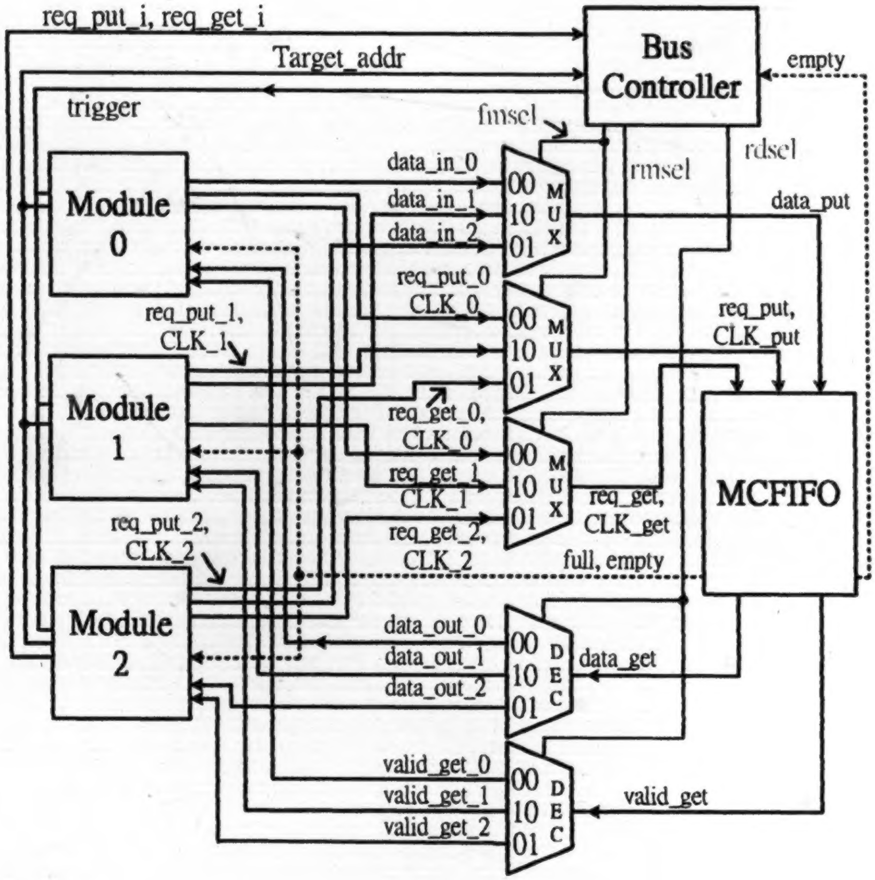

req_put_i, req_get_i
Target_addr
trigger
Bus Controller
empty
fmsel
rmsel
rdsel
data_put
Module 0
Module 1
Module 2
data_in_0
data_in_1
data_in_2
00
10
01
MUX
req_put_0
CLK_0
req_put_1,
CLK_1
req_put_2,
CLK_2
00
10
01
MUX
req_get_0,
CLK_0
req_get_1
CLK_1
req_get_2,
CLK_2
00
10
01
MUX
req_get,
CLK_get
req_put,
CLK_put
MCFIFO
full, empty
data_out_0
data_out_1
data_out_2
00
10
01
DEC
data_get
valid_get_0
valid_get_1
valid_get_2
00
10
01
DEC
valid_get

Figure 4.7: Mixed-Clock Shared Bus

*req_put*=0, *empty*=1 and *req_get*=0. The MCFIFO informs all the modules of its global state and assert *full* or *empty* according to the condition of the MCFIFO. The MCFIFO stalls any put or get operation, if its *full* or *empty* is asserted, respectively. The initiator has to maintain the data items to be sent if the MCFIFO is full or the data is lost.

## 4.4 Summary

In this chapter, we describe the use of MCFIFO in providing multi-point interconnection in SOCs. The goal is to provide interconnection between SoC modules with a smaller area overhead than using MCFIFOs and configure them in a mesh interconnection network. We first explore data broadcasting using the combination of MCFIFOs and shift registers. We then expand the concept into a data broadcasting shared bus, in which we employ the same shift registers to manipulate the data size and data rate according to the clock ratio between the modules and the MCFIFO. The modules in the DBFIFO and DBSB are required to operate at pre-determined rational clock rates for correct operation. We further enhance the design by presenting a highly robust and flexible mixed-clock shared bus architecture. The MCSB allows the modules to operate at true multiple clock frequencies. The MCFIFOs in the MCSB run at the clock rates provided by clock signals from the initiators and targets. As a result, we can fully utilize the mixed-clock interface capability of the MCFIFO. In the next chapter, we will evaluate the performance of the different interconnection designs presented in this chapter.

# Chapter 5

# Design and Performance Evaluation

The chapter is divided into three sections: Test Models of mixed-clock buses, Performance Analysis, and Performance comparison with common synchronous SoC Buses.

Section 5.1, Test Models of the mixed-clock buses discusses the method that is used to verify the two bus designs. Section 5.2, performance analysis, presents experimental results of the two shared buses. Section 5.3 provides a summary of the performance of other commonly used SoC bus architectures.

## 5.1 Test Models of Mixed-clock Buses

Verilog test models of the MCFIFO, DBSB and MCSB are generated for verifying the design and analyzing their performance. The test model of the DBSB contains two initiators and two targets, as configured in Figure B.1. The MCSB test model contains three modules. Each of them can be an initiator or a target, as shown in Figure 4.7. When one module acts as a initiator, any of the other two can become target.

The two test models are built such that initiators request, as soon as possible, another data transfer upon the completion of a previous one. This is achieved by allowing one initiator to continuously put data onto the bus for many *CLK_put* cycle or until the bus is full. This allows the model to exercise the timing of the interfaces more rigorously than should occur in practice.

Each Verilog model is verified in simulations using Cadence *VerilogXL*. The models provide a realization of the FIFO behaviors under various operating conditions and help to verify their functionalities. Three test cases are designed to simulate bus usage of the DBSB in an SoC. The first case exercises a scenario where a single initiator reads data from a single target. In practice, it can be a situation where a processor fetches instructions from on-chip memory in an SoC. This test involves activity in Module 0 and Module 2 of Figure B.1. In this case, Module 0, which has the highest bus access priority, acts as the initiator and Module 2 as the target. Module 0 requests access to the DBSB and asks Module 2 to receive data from the bus. Module 2 responds by asserting *req_get_1* and then retrieves data from the bus.

In the second case, we simulate a scenario in which one initiator sends data to two targets. In this case, the initiator broadcasts data to two targets operating at different speeds. This test involves activity in Module 0, Module 2 and Module 3 of Figure B.1. In this case, Module 0 requests to push data onto the DBSB and asks both Module 2 and Module 3 to receive it.

The final case involves two initiators accessing two different targets. This case attempts to saturate the DBSB bus and simulates mutually exclusive bus use. It involves activities between Module 0 and Module 2, and Module 1 and Module 3 of Figure B.1. First, Module 0 acts as the initiator and attempts to push data to Module 2. After that, Module 0 hands over access to Module 1. Module 1 requests the access of the DBSB bus and pushes data to Module 3.

For MCSB, the second test case does not exist since the MCSB does not support broadcasting of data. Two test cases are therefore established for verifying the

MCSB design. In the first test case, one initiator sends data to a target. This case involves activity in Module 0 and Module 2 of Figure 4.7. Module 0 acts as the initiator in this case and tries to send data to Module 2. In the second test, all three modules in Figure 4.7 take turns to act as initiators and targets to test mutually exclusive bus uses. First, Module 0 is assigned to be the initiator and Module 1 to be the target. After several data transfers from Module 0 to Module 1, the bus controller grants bus access to Module 1 for sending data to Module 2. Then, Module 2 becomes the initiator and sends data to Module 0. These test procedures are repeated to verify the proper functionality of the MCSB. For both DBSB and MCSB test models, two versions of the models are made using a four-place (four FIFO cells) and an eight-place MCFIFOs as the building blocks. By using these models, the functionalities of the two shared buses are verified. Waveforms of parts of the simulations involving four-place DBSB and MCSB are presented in Appendix B.

Table 5.1: DBSB Functional Test Cases

| Test Case | Description |
|---|---|
| Case 1 | Module 0 sends data to Module 2 |
| Case 2 | Module 0 broadcasts data to Module 2 and 3 |
| Case 3 | Module 0 sends data to Module 2<br>then Module 1 sends data to Module 3 |

## 5.2 Performance Analysis

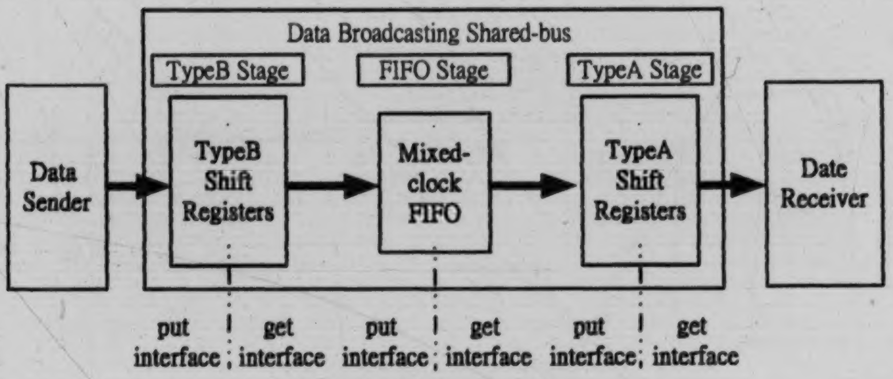
To improve simulation accuracy, the models are synthesized using Synopsys library components and simulated using Synopsys *Design Analyzer* in 0.18*um* CMOS technology, at 1.6V and 300K. All simulations are pre-layout. The results of the maximum throughput and latency, and comparisons between the MCFIFO, DB-FIFO, DBSB and MCSB are summarized in Table 5.2. The shared-bus designs

Table 5.2: Performance Results

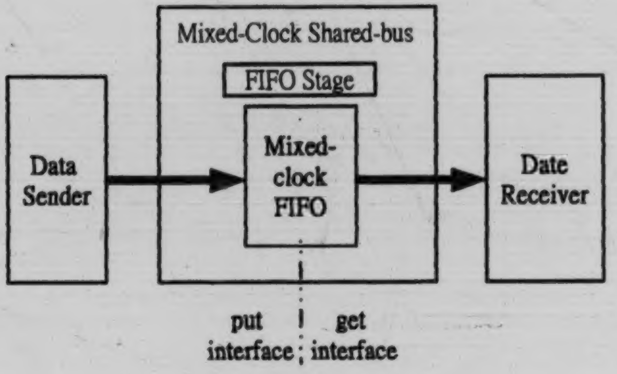
| **Bus Type** | **Stage** | **Throughput** (MHz) | | | | **Latency** (ns) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 4-place | | 8-place | | 4-place | | 8-place | |
| | | put | get | put | get | Min | Max | Min | Max |
| MCFIFO | FIFO | 685 | 625 | 658 | 602 | 3.29 | 4.38 | 4.09 | 4.78 |
| DBFIFO | FIFO | 685 | 625 | 658 | 602 | 5.84 | 7.21 | 5.96 | 7.49 |
| | Data Broadcasting | 730 | 714 | 730 | 714 | | | | |
| DBSB | Type-B | 667 | 667 | 667 | 667 | 7.14 | 8.13 | 7.29 | 8.33 |
| | FIFO | 454 | 450 | 437 | 416 | | | | |
| | Type-A | 730 | 714 | 730 | 714 | | | | |
| MCSB | FIFO | 493 | 485 | 476 | 454 | 4.06 | 5.41 | 4.31 | 5.71 |

are capable of interface between two or more different clock domains. To obtain the throughput of the buses, timing analysis is performed at the put and the get interfaces of various stages of the buses. Each interface corresponds to a single clock domain. For example, since data transfer through MCFIFO crosses two clock domains, the throughput of a MCFIFO contains two parts: the inverse of the cycle time for a put operation and the inverse of the cycle time for a get operation. As a result, when we perform timing analysis using *Design Analyzer*, the MCFIFO circuit is first divided into two sections, synthesized and then analyzed. This is similar to dividing the DBFIFO into the FIFO stage and the data broadcasting stage with each stage having its own put and get interface in Figure 4.1.

For DBSB, with the addition of two shift registers to each ends of the embedded MCFIFO, we have to divide the design into three separate stages, as shown in Figure 5.1, to obtain more accurate experimental results. The throughput of the buses is analyzed at each interface. The results for maximum throughput are expressed as the maximum clock frequency with which that interface can be clocked. From Table 5.2, it is obvious that some stages provide much greater throughput than others. However, the overall throughput of a bus is limited by the stage with the lowest throughput. Latency is the delay for a data to travel from the input of one reference


Data Broadcasting Shared-bus
TypeB Stage
FIFO Stage
TypeA Stage
Data Sender
TypeB Shift Registers
Mixed-clock FIFO
TypeA Shift Registers
Date Receiver
put interface
get interface
put interface
get interface
put interface
get interface

(a) Data Broadcasting Shared-Bus


Mixed-Clock Shared-bus
FIFO Stage
Data Sender
Mixed-clock FIFO
Date Receiver
put interface
get interface

(b) Mixed-clock Shared-bus

Figure 5.1: Synchronous Interfaces of the Shared Buses

point to the output of another one. Therefore, to obtain the latency of the three designs, we synthesize and analyze the designs without dividing them into stages. To calculate the latency of a standalone MCFIFO, a data item is pushed through an empty FIFO. The latency is thus defined as the elapsed time between the moment the data is placed onto *data_put* to the moment when that data appears on *data_get*. For DBSB and MCSB, the experimental setup for latency is as follows: with the bus being empty to begin with, a initiator that has access to the bus requests to send a data item. It places a data item onto the bus. The latency is calculated as the elapsed time between the moment the *data_in_i* bus has valid data to the moment when the data item appears on *data_out_i* of the shared bus. The latency of the MCSB is less than that of the DBSB since the data item does not go through additional shift register stages.

Latency also varies according to when the data items are enqueued by the put interface of the MCFIFO of each bus. If the data item is enqueued by the put interface when *empty* is asserted, the latency is maximized, otherwise, the latency is at minimal. This is because it requires extra time for the two empty detectors to calculate the new state of the MCFIFO before issuing the correct output signals to the get controller. Also, the bus controller contributes to the latency by adding delays through the traffic control signals.

Since both MCSB and DBSB interface with mixed-clock modules through their embedded MCFIFOs, the performance of the MCFIFO directly affects the two shared buses. In normal operations, the MCFIFO has a low chance of experiencing synchronization failure and data loss. It is important to note that once a valid data is enqueued onto a FIFO cell, it can be immediately outputted onto the *data_get* bus on the next *CLK_get*, if the MCFIFO is neither near full nor empty. Therefore, the two bus design can provide low latency data transfer. In this thesis, we define a data transfer delay of less than three transmitter's clock cycles as low latency. The worst-case operation of the MCFIFO occurs when there is a large mismatch in the communication rate (2x or more) between sender and receiver, and the FIFO constantly hits the full or empty state, resulting in a stall in the put or get operation. When *CLK_put* and *CLK_get* are not highly mismatched, using two latches for synchronizing the global control signals are sufficient for good performance. The synchronization of global control signals can be made arbitrarily robust even at

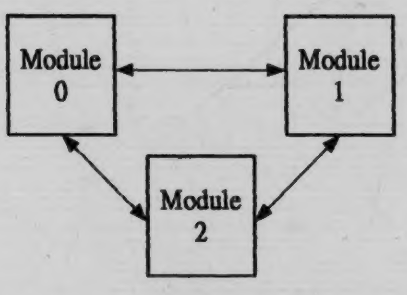
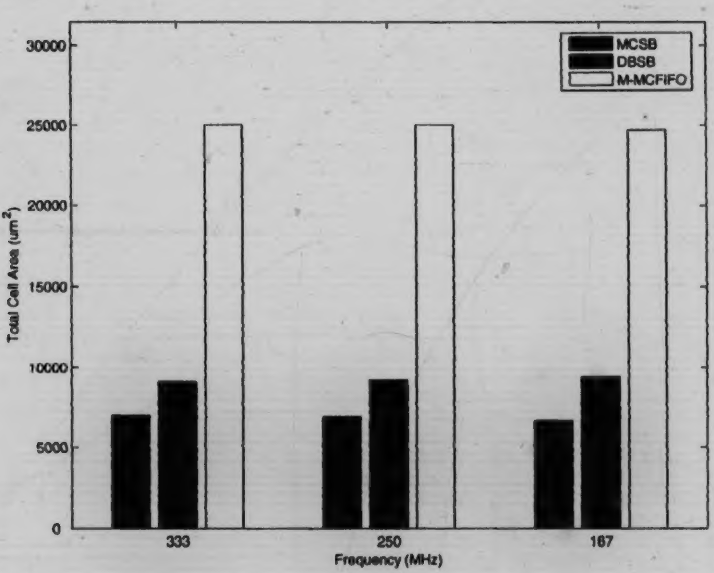


Figure 5.2: Example of SoC Modules Configuration

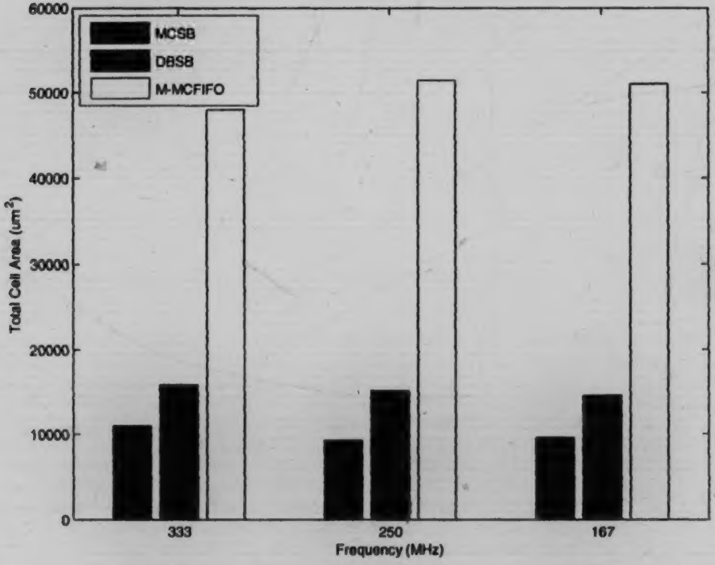
high clock frequencies. At higher clock frequencies, more flip-flops can be added to global control signals if synchronization of these signals does not contribute to a reduction of performance. As the number of synchronizers in the full and the empty detectors get changed, the definitions of *full*, *ne*, and *te* also need to be updated as mentioned in Chapter 3. Otherwise, the detectors may introduce underflow or overflow to the FIFO.

One of the main goals of this thesis is to interconnect multiple modules in an SoC using the MCFIFO but without employing a point-to-point topology. We argued that if the modules are interconnected in a point-to-point fashion for a system configured as shown in Figure 5.2, large amount of resources would have to be allocated to the interconnection network. In the case of Figure 5.2, six MCFIFOs are needed for bidirectional data transfer among the three modules. The MCSB and DBSB, on the other hand, are able to provide mixed-clock interface to multiple modules with just two or less MCFIFOs. To justify this argument, the area and power consumption of the two shared buses with the capability to interconnect to three modules are compared to that of using three separate MCFIFO pairs.

Figure 5.3a and Figure 5.3b illustrate the total cell area occupied by the three design when they are synthesized to meet three different timing constraints, namely, three different maximum clock frequencies. Figure 5.4a and Figure 5.4b show the

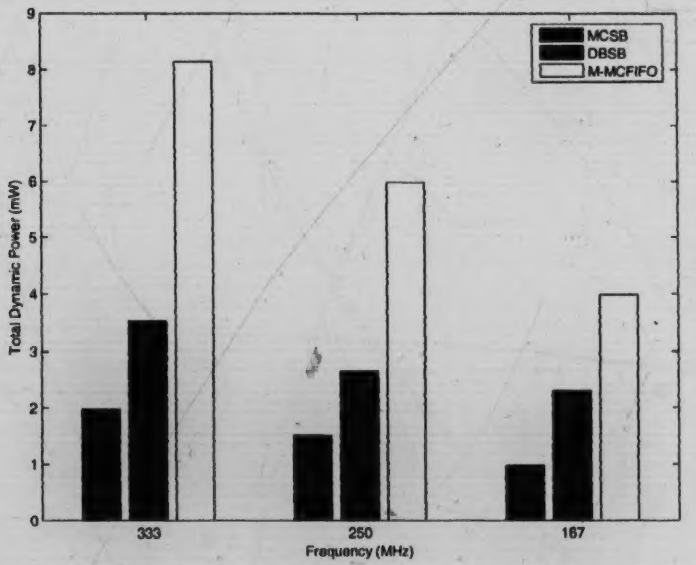


**(a) Implemented using a 4-place MCFIFO**

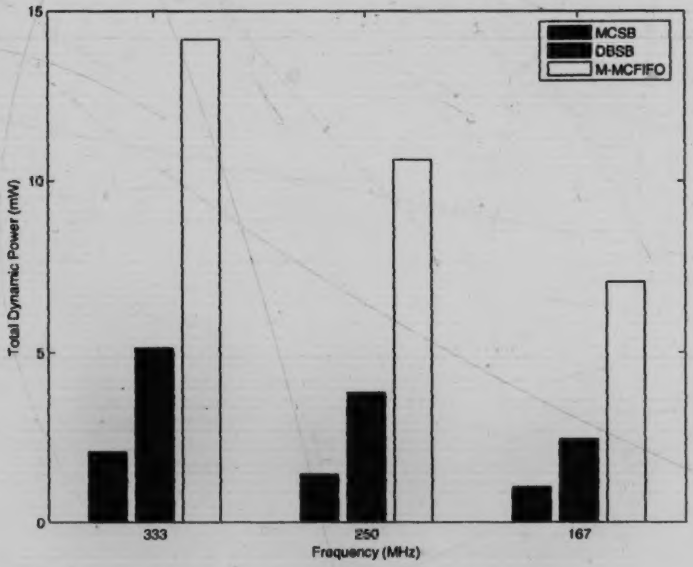


**(b) Implemented using a 8-place MCFIFO**

Figure 5.3: Total Cell Area Usage

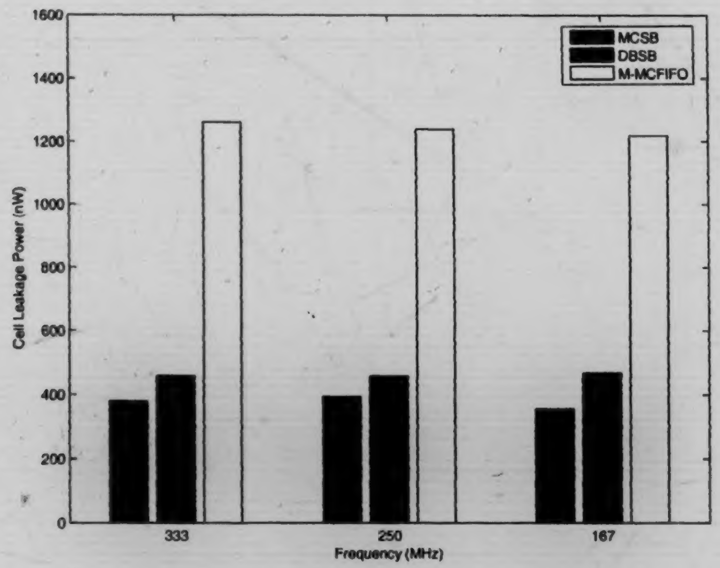


(a) Implemented using a 4-place MCFIFO

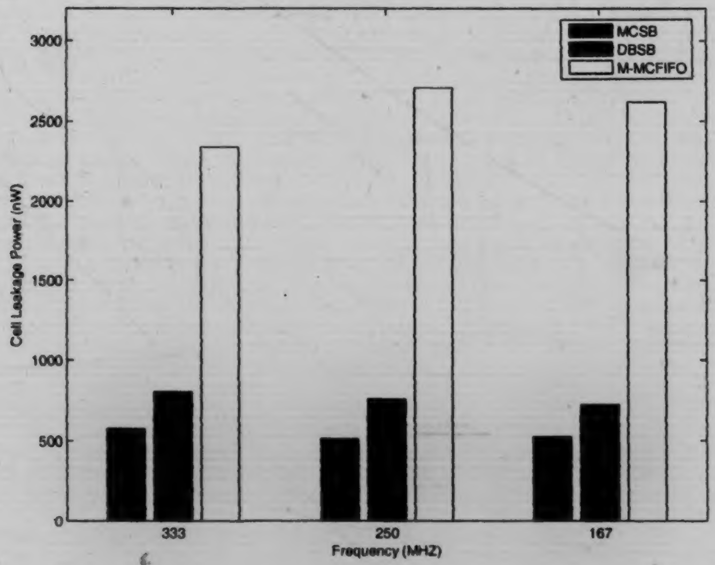


(b) Implemented using a 8-place MCFIFO

Figure 5.4: Total Dynamic Power Consumption



(a) Implemented using a 4-place MCFIFO



(b) Implemented using a 8-place MCFIFO

Figure 5.5: Total Cell Leakage Power

dynamic power consumption of the three designs while Figure 5.5a and Figure 5.5b present the cell leakage power of the designs operating at different frequencies. To implement the on-chip interconnection using multiple MCFIFOs in a point-to-point scheme has shown to be more expensive then using the shared buses, but offers higher data transfer bandwidth. The result indicates overwhelmingly the benefit of utilizing MCSB and DBSB to interconnect modules. Using MCSB and DBSB as the communication network can significantly reduce the cell area and power consumption of the SoC design.

One disadvantage associated with the DBSB is that it may require the data sender to send a group of data items for every data transfer. This happens when a type-A shift register is used to interact with the put or get interface of the MCFIFO. Since a type-A N-to-1 shift register only places data onto its output when it has received N data items, the same amount of data items are needed to be sent on each data transfer, or the bus will stall. However, a type-B shift register require no such limitation.

## 5.3 Comparison with Synchronous Buses

Synchronous SoC buses use the global on-chip clock to regulate the transfer of data between devices. Making a direct performance comparison between on-chip buses is very difficult since each implementation of the buses is different and chips may be fabricated on different processes technologies. In addition, the bus clock specified may not be the maximum possible frequency that the bus is design for, but instead a clock frequency that is derived from the main processor clock. The DBSB and MCSB have only been simulated using conventional CAD tools. Nevertheless, we have provided a list of throughput and latency of other popular SoC buses as a reference in Table 5.3. Here, throughput is expressed as the maximum amount of data

Table 5.3: Performance comparisons of Synchronous Buses [3,4]

| Bus | Bus Width (bits) | Clock Rate (MHz) | Throughput (MB/s) | Latency Min (ns) | Product Implemented On | CMOS feature Size (um) |
|---|---|---|---|---|---|---|
| MCSB | 32 | 83 | 332 | 24 | NA | 0.18 |
| DBSB | 32 | 83 | 332 | 24 | NA | 0.18 |
| AMBA ASB | 32 | 50 | 200 | 40 | Cirrus Logic CL7110 | 0.6 |
| AMBA AHB | 32 | 150 | 600 | 14 | ARM10 | 0.18 |
| CoreConnect OPB | 32 | 50 | 200 | 40 | PowerPC 405GP | 0.25 |
| CoreConnect OPB | 32 | 66 | 264 | 30 | PowerPC 440 Core | 0.18 |
| CoreConnect PLB | 64 | 100 | 800 | 20 | PowerPC 405GP | 0.25 |
| CoreConnect PLB | 128 | 133 | 2128 | 15 | PowerPC 440 Core | 0.18 |

that can be transferred through the bus per second. The result for MCSB and DBSB are obtained by applying two clock signals with the same frequency and phase to the put interfaces and get interfaces of the shared-buses. This effectively transforms the designs into synchronous buses. The latency of the MCSB and DBSB equals two clock periods if they are operated as synchronous buses, which is in par with that of the synchronous buses presented in Table 5.3.

## 5.4 Summary

In this chapter, we discuss how the design of the DBSB and MCSB are verified. The performance of the two shared buses is then presented in terms of maximum throughput and latency of data transfer. We also investigate and compare the benefits of using DBSB or MCSB to provide multiple-point communication for SoC modules over using a point-to-point, multiple MCFIFOs methodology. Finally, we provide a comparison of the performance of the DBSB and MCSB with conventional synchronous SoC on-chip buses.

[ No text ]

# Chapter 6

# Conclusion

As the complexity of SoCs grow, the on-chip buses or interconnect architectures start to dominate system performance. Future SoC devices will require high performance on-chip buses that are sufficiently flexible and robust in order to fulfill the wide variety of tasks. In addition, the physical limits of system scaling and clock frequency of integrated circuit have prompted many engineers to design SoCs in which global synchronization are avoided. This means SoC integrators must employ high performance on-chip buses that can provide efficient interconnection between modules operating with different clock frequencies.

Many mixed-timing on-chip communication schemes have been previously proposed by other research groups. However, most of these are designed to provide separate point-to-point interfaces between mixed-clock modules only. In this thesis, we proposed two new mixed-clock shared-bus architectures that allow efficient communication between modules. The design of the proposed mixed-clock shared-bus architectures are detailed in Chapter 4. The designs rely on the idea of token passing and are based on the design of the low latency MCFIFO presented in [1]. MCFIFO is designed to be suitable for high-bandwidth communication and is highly robust in addressing synchronization issues as discussed in Chapter 2. However, it is initially designed to be use as a point-to-point communication channel. In this thesis, we have expanded the work and converted it into two shared bus designs that enable multi-point interconnection which can be implemented in SoCs.

The MCFIFO has been carefully designed in order to avoid and resolve synchronization failure caused by metastability, FIFO overflow and FIFO underflow. The design complexity of the two shared buses is far less than other completely asynchronous approaches, for example, the MARBLE bus [3]. The two shared buses can be designed and simulated using HDL code and can be partitioned into reusable components. The realization of the design does not require any custom built circuit component or the knowledge of path delay and timing information in advance (unlike many micropipeline designs where delay components must be added to the control path of the design). The advantage of using the two shared buses is that it provides a low-latency, high performance interconnection for multiple modules without the need of implementing multiple MCFIFOs. We have demonstrated in the MCSB that to provide interconnections between three modules operating at different speed, only one MCFIFO is required as opposed to six if a point-to-point topology is employed. This can significantly reduce the area and the power consumption of a design. DBSB provides a unique means to broadcast data to multiple receivers that are operating at rational clock frequencies. On the other hand, MCSB can efficiently interface modules operating at truly multiple clock frequencies.

## 6.1 Future Work

Although the functionality and performance of the two shared buses are verified in this thesis, it has not been physically implemented in a real SoC. We are confident about the two shared bus designs but there are still uncertainties about its behavior and performance in a real SoC environment. It is beneficial to test the design by building an SoC using either of the two shared buses as the communication platform.

As part of future work, the metastability of the shared bus designs can be simulated by forcing the shared buses to enter the MSS. One can also calculate how often the shared buses may encounter an MSS as part of the performance analysis.

Another interesting concept for future work can be the study of the communication between the FIFO cells and the global state detectors. In this work, the communication are established without the use of a synchronizer, e.g. double flip-flops. However, by adding synchronizers between the FIFO cells and the detectors, the chance of synchronization failure of the MCFIFO can be reduced, consequently increases the robustness of the MCFIFO. As a tradeoff, additional synchronizers increase the latency of each transfer to slightly increase the robustness of the MCFIFO operation.

In addition, the use of shift register limits the application of the DBSB since it requires the module to operate at pre-determined rational multiple frequencies. A study of how to improve the shift register is going to assist on enhancing the robustness of the DBSB. Essentially, a new method may have to be researched to enable data broadcasting in multiple clock domains.

The two shared buses are only equipped with a very simple bus controller design. The bus controller is designed to test the most basic functions of the shared buses. It is certain that in a real SoC, a more sophisticated bus arbitrator design is needed to govern the operation of the bus. Also, potential synchronization issues of the bus controller can be investigated. For example, the bus control signals can be synchronized to allow safe bus access transfers between on-chip modules. Therefore, a method for synchronizing the bus control signals to the modules, e.g. two flip-flop synchronizer method, can be studied.

However, the introduction of new components may adversely affect the overall performance of the bus. Design tradeoffs must be made to maximize the performance of the bus.

# Bibliography

[1] T. Chelcea, S.M. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Transactions on Very Large Scale Integration*, vol. 12, no. 8, pp. 857–873, 2004.

[2] T. Chelcea, S.M. Nowick, "A low-latency asynchronous FIFO's using token rings," in *Proceedings of the 6th IEEE International Symposium on Asynchronous Circuits and Systems*, 2000, pp. 210–220.

[3] J. Bainbridge, *Asynchronous System-on-Chip Interconnect*, Springer, 2001.

[4] Advanced RISC Machines Limited (ARM), *AMBA Specification*, ARM Limited, rev 2.0 edition, 1999.

[5] N.H.E.Weste, D.Harris, *CMOS VLSI Design - A Circuits and Systems Perspective*, Addison Wesley, 3rd edition, 2005.

[6] Cadence Digital IC Design White Paper, "Closing the nanometer yield chasm," http://www.cadence.com/whitepapers, 2001.

[7] F. Mu, C. Svensson, "Self-tested self-synchronous circuit for mesochronous clocking," *IEEE Transactions on Circuits and Systems-II*, vol. 48, no. 2, pp. 129–140, 2001.

[8] J. Cong, L. He, C.K. Koh, P.H. Madden, "Performance optimization of VLSI interconnect layout," *Integration, the VLSI Journal (Invited)*, vol. 21, no. 1,2, pp. 1–94, 1996.

[9] P. Ramanathan, A.J. Dupont, K.G. Shin, "Clock distribution in general VLSI circuits," *IEEE Transactions on Circuits and Systems-I*, vol. 41, no. 5, pp. 395–404, 1994.

[10] C.L. Seitz, "System timing," in *Introduction to VLSI Systems*, chapter 7.

Addison-Wesley Publishing Company, 1980.

[11] E. Sutherland, "Micropipelines," *Communication of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[12] J.T. Yantchev, C.G. Huang, M.B. Josephs, I.M. Nedelchev, "Low-latency asynchronous FIFO buffers," in *Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, 1995, pp. 24–31.

[13] Y.S. Kang, K.H. Huh, S. Kang, "New Scan Design of Asynchronous Sequential Circuits," in *Proceedings of the 1st IEEE Asia Pacific Conference on ASICs*, 1999, pp. 355–358.

[14] A. Peeters, K. van Berkel, "Single-rail handshake circuits," in *Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, 1995, pp. 53–62.

[15] T.A. Garcoa, A.J. Acosta, J.M. Mora, J. Ramos, J.L. Huertas, "Self-timed boundary-scan cells for multi-chip module test," in *Proceedings of the 16th IEEE VLSI Test Symposium*, 1998, pp. 92–97.

[16] J. Sparso, J. Staunstrup, M.D. Sorensen, "Design of delay insensitive circuits using multi-ring structures," in *Proceedings of the 1992European Design Automation Conference*, 1992, pp. 15–20.

[17] M. Ferretti, P.A. Beerel, "Single-track asynchronous pipeline templates using 1-of-N encoding," in *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 1008–1015.

[18] W.J. Bainbridge, S.B. Furber, "Delay-insensitive, point-to-point interconnect using 1-of-4 codes," in *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, 2001, pp. 118–126.

[19] W.J. Bainbridge, W.B. Toms, D.A. Edwards, S.B. Furber, "Delay-insensitive, point-to-point interconnect using m-of-n codes," in *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, 2003, pp. 132–140.

[20] T. Hanyu, T. Takahashi, M. Kameyama, "Bidirectional data transfer based asynchronous VLSI system using multiple-valued current mode logic," in

*Proceedings of the 33rd International Symposium on Multiple-Valued Logic*, 2003, pp. 99–104.

[21] S. Furber, D.A Edwards, J.D. Garside, "AMULET3; a 100 MIPS asynchronous embedded processor," in *Proceedings of the 2000 International Conference on Computer Design*, 2000, pp. 329–334.

[22] M. Afghahi, C. Svensson, "Performance of synchronous and asynchronous schemes for VLSI systems," *IEEE Transactions on Computers*, vol. 41, no. 7, pp. 858–872, 1992.

[23] A. Kondratyev, K. Lwin, "Design of asynchronous circuits by synchronous CAD tools," *IEEE Design and Test of Computers*, vol. 19, no. 4, pp. 107–117, 2002.

[24] B. Mesgarzadeh, C. Svensson, A. Alvandpour, "A new mesochronous clocking scheme for synchronization in SoC," in *Proceedings of the 2004 International Symposium on Circuits and Systems*, 2004, vol. 2, pp. II605–II608.

[25] A. Chakraborty, M.R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *Proceedings of the 9th IEEE International Symposium on Asynchronous Circuits and Systems*, 2003, pp. 78–88.

[26] D.M. Chapiro, *Globally-Asynchronous, Locally Synchronous Systems*, Ph.D. thesis, Department of Computer Science, Standford University, 1984, STANCS-84-1026.

[27] J. Mutterbach, T. Villiger, W. Fichtner, "Practical design of globally-asynchronous, locally-synchronous systems," in *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000, pp. 52–59.

[28] K.Y. Yun, R.P. Donohue, "Pausible clocking: a first step toward heterogeneous systems," in *Proceedings of the 1996 International Conference on Computer Design*, 1996, pp. 118–123.

[29] D.S Bormann, P.Y.K. Cheung, "Asynchronous wrapper for heterogeneous systems," in *Proceedings of the 1997 International Conference on Computer Design*, 1997, pp. 307–314.

[30] S. Moore, G. Taylor, "Point to point GALS interconnect," in *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2002, pp. 62–68.

[31] A.J. Winstanley, A. Garivier, M.R. Greenstreet, "An event spacing experiment," in *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2002, pp. 42–51.

[32] W.J. Dally, J.W. Poulton, "Digital systems engineering," in *Synchronization*, chapter 10. Cambridge University Press, 1998.

[33] J.N. Seizovic, "Pipeline synchronization," in *Proceedings of the 1st International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994, pp. 87–96.

[34] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Proceedings of the 9th IEEE International Symposium on Asynchronous Circuits and Systems*, 2003, pp. 89–97.

[35] G.N. Pham, K.C. Schmitt, "A high throughput, asynchronous, dual port FIFO memory implemented in ASIC technology," in *Proceedings of the 2nd Annual International IEEE ASIC Seminar and Exhibition*, 1989, pp. P3 – 1/1–4.

[36] J.H. Jiang, W.B. Jone, S.C. Chang, S. Ghosh, "Embedded core test generation using broadcast test architecture and netlist scrambling," *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 435–443, 2003.

[37] K.J. Lee, J.J. Chen, C.H. Hung, "Using a single input to support multiple scan chains," in *Proceedings of the IEEE International Conference on Computer-Aided Design 1998*, 1998, pp. 74–78.

[38] American National Standards Institute (ANSI), *Small Computer System Interface (SCSI)*, ANSI, 1986.

# Appendix A

# Verilog Code

## A.1 Synthesizable Verilog Model of DBSB (using a 4-place MCFIFO)

```
//---------------------------------------------------------------------
//
//      This confidential and proprietary software may be used only
//     as authorized by the University of Alberta
//     In the event of publication, the following notice is applicable:
//
//                  (C) COPYRIGHT 2005  University of Alberta
//                         ALL RIGHTS RESERVED
//
//      The entire notice above must be reproduced on all authorized
//     copies.
//
// AUTHOR:       Edmund Fung                    LAST REVISION: May 26, 2005
//
// VERSION:      Simulation Architecture
//
//---------------------------------------------------------------------
//---------------------------------------------------------------------
//
// Toplevel Module of SoC with DBSB
//
//---------------------------------------------------------------------
//
module toplevel_DBSB_SoC (rst,CLK_0,CLK_1,CLK_2,CLK_3,f_CLK_put,f_CLK_get,
            r_CLK_put,r_CLK_get,f_ptok_control,r_ptok_control,
            f_gtok_control,r_gtok_control,go_0,go_1,go_2,go_3,
            f_en_get,r_en_get,req_put_0,req_put_1,req_put_2,
            req_put_3,int_addr);

  parameter  int_bus_width    = 4;
  parameter  sel_width        = 2;
  parameter  target_addr_width   = 4;
  parameter  req_put_width    = 4;
  parameter  trigger_width    = 4;

  parameter  top_width_0 = (4*int_bus_width);
  parameter  top_width_1 = (2*int_bus_width);
  parameter  top_width_2 = (2*int_bus_width);
  parameter  top_width_3 = (1*int_bus_width);
  input   rst ,CLK_0,CLK_1,CLK_2,CLK_3;
  input   go_0 , go_1 , go_2 , go_3 ;
  input   f_CLK_put , f_CLK_get , r_CLK_put , r_CLK_get ;
  input   f_ptok_control , r_ptok_control , f_gtok_control , r_gtok_control ;

  input   [(trigger_width - 1):0] int_addr;

  output  f_en_get , r_en_get , req_put_0 , req_put_1 , req_put_2 , req_put_3 ;

  wire    req_put_0 , req_put_1 , req_put_2 , req_put_3 ;
  wire    req_get_0 , req_get_1 , req_get_2 , req_get_3 ;
  wire    f_valid_get_0 , f_valid_get_1 , r_valid_get_0 , r_valid_get_1 ;
  wire    [(top_width_0 - 1):0]   data_in_0 ;
  wire    [(top_width_1 - 1):0]   data_in_1 ;
  wire    [(top_width_2 - 1):0]   data_in_2 ;
  wire    [(top_width_3 - 1):0]   data_in_3 ;
  wire    [(top_width_0 - 1):0]   data_out_0 ;
  wire    [(top_width_1 - 1):0]   data_out_1 ;
  wire    [(top_width_2 - 1):0]   data_out_2 ;
  wire    [(top_width_3 - 1):0]   data_out_3 ;
  wire    [(target_addr_width - 1):0] target_addr ,go;
  wire    [(target_addr_width - 1):0] target_addr_0 , target_addr_1 ;
  wire    [(target_addr_width - 1):0] target_addr_2 , target_addr_3 ;
  wire    [(trigger_width - 1):0] trigger ;
  wire    full_0 , empty_0 , full_1 , empty_1 ;

  SoC_module   #(top_width_0 , target_addr_width)
       mod_0     (rst ,CLK_0, req_put_0 , req_get_0 , data_out_0 , data_in_0 ,
         r_valid_get_0 , trigger[3], full_0 ,empty_1 , go_0 , int_addr ,
         target_addr_0 );
  SoC_module   #(top_width_1 , target_addr_width)
       mod_1     (rst ,CLK_1, req_put_1 , req_get_1 , data_out_1 , data_in_1 ,
         r_valid_get_1 , trigger[2], full_0 ,empty_1 , go_1 , int_addr ,
         target_addr_1 );
  SoC_module   #(top_width_2 , target_addr_width)
       mod_2     (rst ,CLK_2, req_put_2 , req_get_2 , data_out_2 , data_in_2 ,
         f_valid_get_0 , trigger[1], full_1 ,empty_0 , go_2 , int_addr ,
         target_addr_2 );
  SoC_module   #(top_width_3 , target_addr_width)
       mod_3     (rst ,CLK_3, req_put_3 , req_get_3 , data_out_3 , data_in_3 ,
         f_valid_get_1 , trigger[0], full_1 ,empty_0 , go_3 , int_addr ,
         target_addr_3 );

  toplevel_DBSB #(int_bus_width , sel_width , target_addr_width ,
         req_put_width , trigger_width)
     DBSB_0  (rst , req_put_0 , req_put_1 , req_get_0 , req_get_1 , req_put_2 ,
         req_put_3 , req_get_2 , req_get_3 ,CLK_0,CLK_1,CLK_2,CLK_3,
         f_ptok_control , f_gtok_control , r_ptok_control ,
         r_gtok_control , f_CLK_put , f_CLK_get , r_CLK_put , r_CLK_get ,
         target_addr , data_in_0 , data_in_1 , data_in_2 , data_in_3 ,
         data_out_0 , data_out_1 , data_out_2 , data_out_3 , full_0 ,
         empty_0 , full_1 , empty_1 , f_valid_get_0 , f_valid_get_1 ,
         r_valid_get_0 , r_valid_get_1 , f_en_get , r_en_get , trigger , go);

  assign  target_addr = (target_addr_0 | target_addr_1 | target_addr_2
                         | target_addr_3);
  assign  go = {go_3 , go_2 , go_1 , go_0};

endmodule
```

```
//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Toplevel Module of DBSB
//
//------------------------------------------------------------------------
//
module toplevel_DBSB (rst, f_req_put_0, f_req_put_1, r_req_get_0, r_req_get_1,
               r_req_put_0, r_req_put_1, f_req_get_0, f_req_get_1, CLK_0,
               CLK_1, CLK_2, CLK_3, f_ptok_control, f_gtok_control,
               r_ptok_control, r_gtok_control, f_CLK_put, f_CLK_get,
               r_CLK_put, r_CLK_get, target_addr, data_in_0, data_in_1,
               data_in_2, data_in_3, data_out_0, data_out_1, data_out_2,
               data_out_3, f_full, f_empty, r_full, r_empty, f_valid_get_0,
               f_valid_get_1, r_valid_get_0, r_valid_get_1, f_en_get,
               r_en_get, trigger, go);

  parameter  int_bus_width   = 6;
  parameter  sel_width       = 2;
  parameter  target_addr_width  = 4;
  parameter  req_put_width   = 4;
  parameter  trigger_width   = 4;

  parameter  valid_state     = 1'b0;

  parameter  addr_width  = target_addr_width + req_put_width;
  parameter  top_width_0 = (4*int_bus_width);
  parameter  top_width_1 = (2*int_bus_width);
  parameter  top_width_2 = (2*int_bus_width);
  parameter  top_width_3 = (1*int_bus_width);

  input  rst, f_CLK_put, r_CLK_put;
  input  f_CLK_get, r_CLK_get;
  input  f_req_put_0, f_req_put_1, f_req_get_0, f_req_get_1;
  input  r_req_put_0, r_req_put_1, r_req_get_0, r_req_get_1;
  input  CLK_0, CLK_1, CLK_2, CLK_3;
  input  f_ptok_control, f_gtok_control, r_ptok_control, r_gtok_control;

  input  [(target_addr_width - 1):0] go;
  input  [(req_put_width - 1):0] target_addr;
  input  [(top_width_0 - 1):0]   data_in_0;
  input  [(top_width_1 - 1):0]   data_in_1;
  input  [(top_width_2 - 1):0]   data_in_2;
  input  [(top_width_3 - 1):0]   data_in_3;

  output f_valid_get_0, f_valid_get_1, r_valid_get_0, r_valid_get_1;
  output f_full, f_empty, r_full, r_empty;
  output f_en_get, r_en_get;

  output [(trigger_width - 1):0] trigger;
  output [(top_width_0 - 1):0]   data_out_0;
  output [(top_width_1 - 1):0]   data_out_1;
  output [(top_width_2 - 1):0]   data_out_2;
  output [(top_width_3 - 1):0]   data_out_3;

  wire   [(int_bus_width - 1):0] f_data_put;
  wire   [(int_bus_width - 1):0] f_data_get;
  wire   [(int_bus_width - 1):0] f_data_in_0;
  wire   [(int_bus_width - 1):0] f_data_in_1;
  wire   [(int_bus_width - 1):0] f_data_out_0;
  wire   [(int_bus_width - 1):0] f_data_out_1;
  wire   [(int_bus_width - 1):0] r_data_put;
  wire   [(int_bus_width - 1):0] r_data_get;
  wire   [(int_bus_width - 1):0] r_data_in_0;
  wire   [(int_bus_width - 1):0] r_data_in_1;
  wire   [(int_bus_width - 1):0] r_data_out_0;
wire    [(int_bus_width - 1):0] r_data_out_1;

wire    fmsel, rmsel;
wire    [(sel_width - 1):0] fdsel, rdsel;
wire    f_req_put, f_req_get, r_req_put, r_req_get;
wire    f_valid_get, r_valid_get;
wire    [(addr_width - 1):0] address;
wire    [(trigger_width - 1):0] trigger;

assign  data_out_3  = f_data_out_1;
assign  r_data_in_1 = data_in_3;

//forward controls
mux2to1     #(int_bus_width)
      fm0  (rst, f_data_in_0, f_data_in_1, fmsel, f_data_put);
mux2to1     fmc0 (rst, f_req_put_0, f_req_put_1, fmsel, f_req_put);
or          or_0 (f_req_get, f_req_get_0, f_req_get_1);
demux1to2   #(int_bus_width)
      fdm0 (rst, f_data_get, fdsel, f_data_out_0, f_data_out_1);
demux1to2   #(1, sel_width, valid_state)
      fdc0 (rst, f_valid_get, fdsel, f_valid_get_0, f_valid_get_1);

//reverse controls
mux2to1     #(int_bus_width)
      rm0  (rst, r_data_in_0, r_data_in_1, rmsel, r_data_put);
mux2to1     rmc0 (rst, r_req_put_0, r_req_put_1, rmsel, r_req_put);
or          or_1 (r_req_get, r_req_get_0, r_req_get_1);
dec1to2     #(int_bus_width)
      rdm0 (rst, r_data_get, rdsel, r_data_out_0, r_data_out_1);
dec1to2     #(1, sel_width, valid_state)
      rdc0 (rst, r_valid_get, rdsel, r_valid_get_0, r_valid_get_1);


toplevel_MCFIFO   #(int_bus_width)
        MCFIFO_0     (rst, f_full, f_empty, f_req_put, f_data_put,
             f_CLK_put, f_ptok_control, f_CLK_get,
             f_data_get, f_req_get, f_valid_get,
             f_gtok_control, f_en_get);

toplevel_MCFIFO   #(int_bus_width)
        MCFIFO_1     (rst, r_full, r_empty, r_req_put, r_data_put,
             r_CLK_put, r_ptok_control, r_CLK_get,
             r_data_get, r_req_get, r_valid_get,
             r_gtok_control, r_en_get);


DBSB_bus_controller    #(target_addr_width, trigger_width, sel_width)
        bctrl0 (rst, go, target_addr, trigger, fmsel, fdsel, rmsel, rdsel);


//core 0
TypeB_shreg1to4   #(top_width_0, int_bus_width, 3, 3'b100)
   shreg_0f    (rst, f_CLK_put, data_in_0, f_req_put_0, f_data_in_0);

TypeA_shreg4to1   #(int_bus_width, top_width_0)
   shreg_0r    (rst, r_data_out_0, data_out_0, r_CLK_get,
         CLK_0, r_valid_get_0);

//core 1
TypeB_shreg1to2   #(top_width_1, int_bus_width)
   shreg_1f    (rst, CLK_0, data_in_1, f_req_put_1, f_data_in_1);

TypeA_shreg2to1   #(int_bus_width, top_width_1)
   shreg_1r    (rst, r_data_out_1, data_out_1, r_CLK_get,
         CLK_1, r_valid_get_1);
```

```
  //core 2
  TypeB_shreg1to2  #( top_width_2 , int_bus_width )
     shreg_2r   ( rst ,CLK_2, data_in_2 , r_req_put_0 , r_data_in_0 );


  TypeA_shreg2to1  #( int_bus_width , top_width_2 )
     shreg_2f   ( rst , f_data_out_0 , data_out_2 , f_CLK_get ,
            CLK_2, f_valid_get_0 );


endmodule

//------------------------------------------------------------------
//------------------------------------------------------------------
//
// 2 - to - 1 Multiplexer
//
//------------------------------------------------------------------
//
module mux2to1 ( rst ,A,B, sel ,Z);

  parameter width = 1;
  parameter sel_width = 1;

  input  rst;
  input  [( width  - 1):0] A,B;
  input  [( sel_width  - 1):0] sel;
  output [( width  - 1):0] Z;

  reg    [( width  - 1):0] Z;

  always @( sel  or  A  or  B)
    begin
    if ( sel == 0)
      begin
        Z = A;
      end
    else
      begin
        Z = B;
      end
    end

endmodule

//------------------------------------------------------------------
//------------------------------------------------------------------
//
// 1 - to - 2 Decoder
//
//------------------------------------------------------------------
//
module dec1to2 ( rst ,A, sel , Out1 , Out2 );

  parameter width      = 1;
  parameter sel_width  = 2;

  //parameter  tri_state = 'bz;
  parameter  tri_state = {( width ){1 'bz}};

  input  rst;
  input  [( width  - 1):0] A;
  input  [( sel_width  - 1):0] sel;
  output [( width  - 1):0] Out1 , Out2;
  reg    [(width - 1):0] Out1 , Out2;

  always @(A or sel)
    begin
        if ( sel == 2'b10)
      begin
        Out1 = A;
        Out2 = tri_state;
      end
    else
      begin
        if ( sel == 2'b01)
          begin
            Out1 = tri_state;
            Out2 = A;
          end
        else
          begin
            if ( sel == 2'b11)
              begin
                Out1 = A;
                Out2 = A;
              end
          end
      end
    end

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// DBSB Bus Controller
//
//------------------------------------------------------------------------
//
module DBSB_bus_controller (rst,go,target_addr,trigger,fmsel,fdsel,rmsel,rdsel);

  parameter addr_width      = 4;
  parameter trigger_width   = 4;
  parameter sel_width       = 2;
  parameter out_width       = (3*sel_width);

  input  rst;
  input  [(addr_width - 1):0]     go;
  input  [(addr_width - 1):0]     target_addr;

  output fmsel,rmsel;
  output [(trigger_width - 1):0] trigger;
  output [(sel_width - 1):0]      fdsel,rdsel;

  wire   fmsel,rmsel;
  wire   [(sel_width - 1):0]      fdsel,rdsel;
  wire   [(trigger_width - 1):0] trigger;
  wire   [(2*trigger_width - 1):0] address;
  reg    [(out_width - 1):0]      controller_out;


  assign fmsel   = controller_out[5];
  assign fdsel   = {controller_out[4],controller_out[3]};
  assign rmsel   = controller_out[2];
  assign rdsel   = {controller_out[1],controller_out[0]};

  assign address = {go, target_addr};
  assign trigger = target_addr;
  always @(address)
    begin
    case(address)
      8'b00010010 : controller_out = 6'b010000; // Module 0
      8'b00010001 : controller_out = 6'b001000;
      8'b00010011 : controller_out = 6'b011000;
      8'b00100010 : controller_out = 6'b110000; // Module 1
      8'b00100001 : controller_out = 6'b101000;
      8'b00100011 : controller_out = 6'b111000;
      8'b01001000 : controller_out = 6'b000010; // Module 2
      8'b01000100 : controller_out = 6'b000001;
      8'b01001100 : controller_out = 6'b000011;
      8'b10001000 : controller_out = 6'b000110; // Module 3
      8'b10000100 : controller_out = 6'b000101;
      8'b10001100 : controller_out = 6'b000111;
      default : controller_out = 6'b000000;
    endcase
    end

endmodule

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// Type A Shift Register (ratio: 4 input segements into 1 output)
//
//---------------------------------------------------------------------------
//
module TypeA_shreg4to1 (rst,data_in,data_out,CLK_in,CLK_out,enable);

  parameter  in_val_width = 1;
  parameter out_val_width = 4;
  parameter     clk_ratio = (out_val_width/in_val_width);

  input  [(in_val_width - 1):0] data_in;
  input  rst, CLK_in, CLK_out, enable;
  output [(out_val_width - 1):0] data_out;

  wire   [(in_val_width - 1):0]  temp_out_0;
  wire   [(in_val_width - 1):0]  temp_out_1;
  wire   [(in_val_width - 1):0]  temp_out_2;
  wire   [(in_val_width - 1):0]  temp_out_3;

  wire   [(in_val_width - 1):0]  shift_out_0;
  wire   [(in_val_width - 1):0]  shift_out_1;
  wire   [(in_val_width - 1):0]  shift_out_2;
  wire   [(in_val_width - 1):0]  shift_out_3;

  wire   out_en, enable;

  assign data_out = {temp_out_3, temp_out_2 ,temp_out_1, temp_out_0};


  TypeA_shreg_cell #(in_val_width)
        shreg0  (rst, data_in, temp_out_0, shift_out_0,
                     CLK_in, CLK_out, enable, out_en);

  TypeA_shreg_cell #(in_val_width)
        shreg1  (rst, shift_out_0, temp_out_1, shift_out_1,
                     CLK_in, CLK_out, enable, out_en);

  TypeA_shreg_cell #(in_val_width)
        shreg2  (rst, shift_out_1, temp_out_2, shift_out_2,
                     CLK_in, CLK_out, enable, out_en);

  TypeA_shreg_cell #(in_val_width)
```

```
        shreg3  (rst , shift_out_2 , temp_out_3 , shift_out_3 ,
                     CLK_in , CLK_out , enable , out_en );


   data_counter_4    d4_0 ( rst , out_en , CLK_in , enable );


endmodule

//--------------------------------------------------------------------
//--------------------------------------------------------------------
//
// Type A Shift Register ( ratio : 2 input segments into 1 output)
//
//--------------------------------------------------------------------
//
module TypeA_shreg2to1 ( rst , data_in , data_out , CLK_in , CLK_out , valid_get );

   parameter   in_val_width = 1;
   parameter  out_val_width = 2;
   parameter       clk_ratio = ( out_val_width / in_val_width );

   input   [( in_val_width  - 1):0]  data_in;
   input   rst , CLK_in , CLK_out , valid_get;
   output  [( out_val_width  - 1):0] data_out;

   wire    [( in_val_width  - 1):0]  temp_out_0;
   wire    [( in_val_width  - 1):0]  temp_out_1;
   wire    [( in_val_width  - 1):0]  shift_out_0;
   wire    [( in_val_width  - 1):0]  shift_out_1;

   wire    out_en , valid_get;

   assign  data_out = { temp_out_0 , temp_out_1 };



   TypeA_shreg_cell #( in_val_width )
         shreg0  ( rst , data_in , temp_out_0 , shift_out_0 ,
                      CLK_in , CLK_out , valid_get , out_en );

   TypeA_shreg_cell #( in_val_width )
         shreg1  ( rst , shift_out_0 , temp_out_1 , shift_out_1 ,
                      CLK_in , CLK_out , valid_get , out_en );


   data_counter_2  d2_0 ( rst , out_en , CLK_in , valid_get );


endmodule

//--------------------------------------------------------------------
//--------------------------------------------------------------------
//
// Type A Shift Register Cell
//
//--------------------------------------------------------------------

module TypeA_shreg_cell ( rst , in_value , out_value , shift_out ,
             CLK_in , CLK_out , enable , out_en );

   parameter  in_val_width = 1;

   input  [( in_val_width  - 1):0]  in_value;
```

```
  input  rst , CLK_in , CLK_out , enable , out_en ;
  output [(in_val_width - 1):0]  out_value , shift_out ;

  wire    [(in_val_width - 1):0]  shift_in ;
  wire    [(in_val_width - 1):0]  latch_out ;

  assign shift_out = shift_in ;

  enable_reg #(in_val_width)
          u0 ( rst , CLK_in , enable , shift_in , in_value );

  clk_latch   #(in_val_width)
          u1 ( rst , out_en , latch_out , shift_in );

  register    #(in_val_width)
          u2 ( rst , CLK_out , out_value , latch_out );

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Data Counter ( counts  4)
//
//------------------------------------------------------------------------

module data_counter_4 ( rst , out_latch_en ,CLK, counter_en );

  parameter count_width = 3;
  parameter match = 3'b100;

  input   rst ,CLK, counter_en ;
  output out_latch_en ;

  data_counter #(count_width , match ) d0 ( rst ,CLK, out_latch_en , counter_en );

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Data Counter ( counts 2)
//
//------------------------------------------------------------------------

module data_counter_2 ( rst , out_latch_en ,CLK, counter_en );

  parameter count_width = 2;
  parameter match = 2'b10;

  input   rst ,CLK, counter_en ;
  output out_latch_en ;

  counter #(count_width , match ) d0 ( rst ,CLK, out_latch_en , counter_en );

endmodule
```

```
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// Counter
//
//--------------------------------------------------------------------------

module counter ( rst ,CLK, out_count , data_en );

  parameter clk_ratio = 1;
  parameter count_width = (clk_ratio/2);


  input   rst , data_en ,CLK;
  output  out_count ;

  reg  out_count;
  reg  [(count_width - 1):0] count;
  reg  int_rst;

  always @(data_en or count or rst)
    begin
    if ( rst == 1)
      begin
            count <= 2'b00;
      end
    else
      begin
        if ( data_en == 1)
          begin
            count <= count + 2'b01;
          end
          end
    end

  always @(count or rst)
    begin
    if ( rst == 1)
      begin
        out_count = 0;
      end
    else
      begin
        if ( count == 2'b11)
          begin
            out_count = 1;
            count     = 2'b00;
          end
        else
          begin
            out_count = 0;
          end
      end
    end

endmodule
```

```
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// Type B Shift Register (ratio: 1 input into 4 output segments)
//
//--------------------------------------------------------------------------
//
module TypeB_shreg1to4 (rst,CLK_in,data_in,enable,shreg_data_out);

  parameter in_width    = 28;
  parameter out_width   = 7;

  parameter count_width = 2;
  parameter count_max   = 3'b100;
  parameter clk_ratio   = in_width/out_width;
  parameter segEnd3 = in_width - out_width; // 28 - 7 = 21
  parameter segEnd2 = segEnd3 - out_width;  // 21 - 7 = 14
  parameter segEnd1 = segEnd2 - out_width;  // 14 - 7 = 7
  parameter segEnd0 = segEnd1 - out_width;  //  7 - 7 = 0

  input  rst, CLK_in, enable;
  input  [(in_width - 1):0] data_in;
  output [(out_width - 1):0] shreg_data_out;

  wire   [(out_width - 1):0] shift_in_1, shift_in_2;
  wire   [(out_width - 1):0] segment0, segment1, segment2, segment3;
  wire   [(count_width-1):0] count;
  wire   load_sel;

  assign segment0 = data_in[(segEnd1 - 1):segEnd0];
  assign segment1 = data_in[(segEnd2 - 1):segEnd1];
  assign segment2 = data_in[(segEnd3 - 1):segEnd2];
  assign segment3 = data_in[(in_width - 1):segEnd3];

  TypeB_shreg_cell #(out_width)
          TB1 (rst,CLK_in,load_sel,segment0,segment1,shift_in_1);
  TypeB_shreg_cell #(out_width)
          TB2 (rst,CLK_in,load_sel,shift_in_1,segment2,shift_in_2);
  TypeB_shreg_cell #(out_width)
              TB3 (rst,CLK_in,load_sel,shift_in_2,segment3,shreg_data_out);

  load_counter     #(count_width, count_max)
       lc_0 (rst,CLK_in,enable,count);

  load_controller  #(count_width)
       lctrl_0 (rst,enable,count,load_sel);

endmodule

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// Type B Shift Register (ratio: 1 input into 2 output segments)
//
//--------------------------------------------------------------------------
//
module TypeB_shreg1to2 (rst,CLK_in,data_in,valid_get,shreg_data_out);

  parameter in_width    = 14;
  parameter out_width   = 7;

  parameter count_width = 2;
  parameter count_max   = 2'b10;
  parameter clk_ratio   = in_width/out_width;
  parameter segEnd1 = in_width - out_width; // 14 - 7 = 7
  parameter segEnd0 = segEnd1  - out_width; //  7 - 7 = 0
  input  rst , CLK_in , valid_get ;
  input  [(in_width - 1):0] data_in ;
  output [(out_width - 1):0] shreg_data_out ;

  wire   [(out_width - 1):0] segment0 , segment1 ;
  wire   [(count_width -1):0] count ;
  wire   load_sel ;

  assign segment0 = data_in [(segEnd1 - 1):segEnd0 ];
  assign segment1 = data_in [(in_width - 1):segEnd1 ];

  TypeB_shreg_cell #(out_width)
            TB1 ( rst ,CLK_in , load_sel , segment0 , segment1 , shreg_data_out );

  load_counter #(count_width , count_max )
        lc_0 ( rst ,CLK_in , valid_get ,count );

  load_controller #(count_width)
        lctrl_0 ( rst , valid_get , count , load_sel );

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Type B Shift Register Cell
//
//------------------------------------------------------------------------
//
module TypeB_shreg_cell ( rst ,CLK, load_sel ,input1 ,input2 , out_data );

  parameter segment_width = 7;

  input  [(segment_width - 1):0] input1 , input2 ;
  input  rst , CLK, load_sel ;
  output [(segment_width - 1):0] out_data ;

  wire   [(segment_width - 1):0] to_MUX;

  register #(segment_width )
        r0 ( rst ,CLK,to_MUX, input1 );
  mux2to1  #(segment_width )
        m0 ( rst ,to_MUX, input2 , load_sel , out_data );
endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Load Controller
//
//------------------------------------------------------------------------
//Control when to assert the single load_sel
//
module load_controller ( rst ,enable ,count , load_sel );

  parameter count_width = 2;

  input  rst ,enable ;
  input  [(count_width - 1):0] count ;
  output load_sel ;

  reg  load_sel , oneCLKflag ;
  wire ctrl ;
  assign ctrl = (enable & oneCLKflag);

  always @(ctrl or rst)
    begin
    if (rst == 1)
      begin
        load_sel = 0;
      end
    else
      begin
            if (ctrl == 1)
              begin
            load_sel = 1;
              end
            else
          begin
            load_sel = 0;
          end
      end
    end

  always @(count or rst)
    begin
    if (rst == 1)
      begin
        oneCLKflag = 0;
      end
    else
      begin
        if (count == 2'b01)
          begin
            oneCLKflag = 1;
          end
        else
          begin
            oneCLKflag = 0;
          end
      end
    end

endmodule

//----------------------------------------------------------------------
//----------------------------------------------------------------------
//
// Load Counter
//
//----------------------------------------------------------------------
//counter counts and signals load controller when to assert load_sel
//
module load_counter (rst ,CLK, enable , count);

  parameter count_width = 2;
  parameter count_max = 2'b11;

  input   rst ,CLK, enable;
  output  [(count_width - 1) : 0] count;

  reg  [(count_width - 1):0] count;
  reg  cRst;

  always @(posedge CLK or posedge rst)
    begin
    if (rst == 1)
      begin
```

```
          count <= 1'b0;
        end
      else
        begin
          if (cRst == 0)
            begin
              if (enable == 1)
                begin
                  count <= count + 2'b01;
                end
                end
          else
            begin
              count <= 2'b00;
            end
        end
      end

    always @(count)
      begin
      if (count == count_max)
        begin
          cRst = 1;
        end
      else
        begin
          cRst = 0;
        end
      end

endmodule

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//
// Toplevel Module of the Mixed-Clock FIFO
//
//-----------------------------------------------------------------------------
//
module toplevel_MCFIFO (rst, full, empty, req_put, data_put, CLK_put, ptok_control,
                        CLK_get, data_get, req_get, valid_get, gtok_control, en_get);

  parameter width = 7;

  input                    rst, req_put, CLK_put, CLK_get, req_get;
  input                    ptok_control, gtok_control;
  input  [(width - 1):0]   data_put;
  output [(width - 1):0]   data_get;
  output                   full, empty, valid_get, en_get;
  wire                     en_put, en_get;
  wire                     valid;
  wire                     valid_i;

  wire    [3:0]            cell_full, cell_empty;
  wire                     ne, oe;
  wire    [3:0]            ptok_in, gtok_in, ptok_out, gtok_out;

  assign  ptok_in[1] = ptok_out[0];
  assign  ptok_in[2] = ptok_out[1];
  assign  ptok_in[3] = ptok_out[2];

  assign  gtok_in[1] = gtok_out[0];
  assign  gtok_in[2] = gtok_out[1];
  assign  gtok_in[3] = gtok_out[2];
  full_detector   fd_0  ( rst , cell_empty [0] , cell_empty [1] , cell_empty [2] ,
                          cell_empty [3] , CLK_put , full );

  empty_detector  ed_0  ( rst , cell_full [0] , cell_full [1] , cell_full [2] ,
                          cell_full [3] , CLK_get , en_get , oe , ne );


  put_controller  pc_0  ( full , req_put , en_put );

  get_controller  gc_0  ( req_get , ne , oe , valid_i , en_get , valid , empty );

  en_latch        valid_latch_0  ( rst , valid_i , valid_get , valid );


  fifo_cell       #( width )
        ff_0 ( rst , req_put , data_put , en_put , CLK_put , CLK_get , data_get ,
                    en_get , valid_i , ptok_out [0] , ptok_in [0] , cell_full [0] ,
                    cell_empty [0] , gtok_out [0] , gtok_in [0] );
  fifo_cell       #( width )
        ff_1 ( rst , req_put , data_put , en_put , CLK_put , CLK_get , data_get ,
                    en_get , valid_i , ptok_out [1] , ptok_in [1] , cell_full [1] ,
                    cell_empty [1] ,  gtok_out [1] ,  gtok_in [1] );
  fifo_cell       #( width )
        ff_2 ( rst , req_put , data_put , en_put , CLK_put , CLK_get , data_get ,
                    en_get , valid_i , ptok_out [2] , ptok_in [2] , cell_full [2] ,
                    cell_empty [2] , gtok_out [2] , gtok_in [2] );
  fifo_cell       #( width )
        ff_3 ( rst , req_put , data_put , en_put , CLK_put , CLK_get , data_get ,
                    en_get , valid_i , ptok_out [3] , ptok_in [3] , cell_full [3] ,
                    cell_empty [3] , gtok_out [3] , gtok_in [3] );


  tok_controller  ptc_0( ptok_out [3] , ptok_control , ptok_in [0] );
  tok_controller  gtc_0( gtok_out [3] , gtok_control , gtok_in [0] );

endmodule

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// Full Detector
//
//--------------------------------------------------------------------------


module full_detector ( rst , e_0 , e_1 , e_2 , e_3 , CLK_put , full );
  input  rst , e_0 , e_1 , e_2 , e_3 , CLK_put;
  output full ;

  wire   reg_buf , not_1_out , not_0_out ;
  wire   not_e_0 , not_e_1 , not_e_2 , not_e_3 ;
  wire   A, B, C, D, E, F, pre_out ;

  not  not_0   ( not_e_0 , e_0 );
  not  not_1   ( not_e_1 , e_1 );
  not  not_2   ( not_e_2 , e_2 );
  not  not_3   ( not_e_3 , e_3 );

  or   or2_0   (A, not_e_0 , not_e_1 );
  or   or2_1   (B, not_e_1 , not_e_2 );
  or   or2_2   (C, not_e_2 , not_e_3 );
  or   or2_3   (D, not_e_3 , not_e_0 );

  and  and2_0  (E,A,B);
  and  and2_1  (F,C,D);
  and  and2_2  ( pre_out ,E,F);
  not not_4 (not_0_out , pre_out);
  not not_5 (not_1_out , not_0_out);
  register reg_0 (rst ,CLK_put ,reg_buf ,not_1_out);
  register reg_1 (rst ,CLK_put ,full ,reg_buf);

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Toplevel Module of Empty Detector
//
//-----------------------------------------------------------------------

module empty_detector (rst , f_0 , f_1 , f_2 , f_3 , CLK_get , en_get , oe , ne);
  input  rst , f_0 , f_1 , f_2 , f_3 , CLK_get , en_get;
  output oe , ne;

  normal_empty_detector NED_0 (rst , f_0 , f_1 , f_2 , f_3 , CLK_get , ne);
  true_empty_detector   TED_0 (rst , f_0 , f_1 , f_2 , f_3 , CLK_get , en_get , oe);

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Normal Empty Detector
//
//-----------------------------------------------------------------------

module normal_empty_detector (rst , f_0 , f_1 , f_2 , f_3 , CLK_get , ne);
  input  rst , f_0 , f_1 , f_2 , f_3 , CLK_get;
  output ne;

  parameter  in_width = 1;
  parameter  default_out  = {(in_width){1'b1}};

  wire reg_buf , not_0_out , not_1_out , pre_out;
  wire A, B, C, D, E, F, G, H;

  and and2_0 (A,f_0 ,f_1);
  and and2_1 (B,f_1 ,f_2);
  and and2_2 (C,f_2 ,f_3);
  and and2_3 (D,f_3 ,f_0);

  or  or2_0  (E,A,B);
  or  or2_1  (F,C,D);
  or  or2_2  (G,E,F);

  not not_0  (pre_out ,G);
  not not_1 (not_0_out , pre_out);
  not not_2 (not_1_out , not_0_out);
  register  #(in_width ,default_out)
       reg_0 (rst ,CLK_get ,reg_buf ,not_1_out);
  register  #(in_width ,default_out)
       reg_1 (rst ,CLK_get ,ne ,reg_buf);

endmodule
```

```
//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// True Empty Detector
//
//------------------------------------------------------------------------

module true_empty_detector (rst, f_0, f_1, f_2, f_3, CLK_get, en_get, oe);
  input   rst, f_0, f_1, f_2, f_3, CLK_get, en_get;
  output oe;

  parameter  in_width = 1;
  parameter  default_out  = {(in_width){1'b1}};

  wire    pre_out, A, B, C;
  wire    reg_buf, prev_oe;

         or OR2_A (A,f_0,f_1);
     or OR2_B (B,f_2,f_3);
     or OR2_C (C,A,B);
    not not_1 (pre_out,C);

    register #(in_width,default_out)
         reg_0 (rst,CLK_get,reg_buf,pre_out);
          or OR2_0 (prev_oe,reg_buf,en_get);
    register #(in_width,default_out)
         reg_1 (rst,CLK_get,oe,prev_oe);

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Put Controller
//
//------------------------------------------------------------------------

module put_controller (full, req_put, en_put);
  input   full, req_put;
  output en_put;

  reg nfull, en_put;

  always @(req_put or full)
    begin
    nfull = !(full);
    en_put = req_put & nfull;
    end

endmodule
```

```
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Get Controller
//
//------------------------------------------------------------------------

module get_controller (req_get, ne, oe, valid_i, en_get, valid_get, empty);
  input   req_get, ne, oe, valid_i;
  output  en_get, valid_get, empty;

  wire    int1, int2, int3;
  assign  int1 = valid_i;

  and  AND2_0(empty, ne, oe);
  not  NOT_0(int2, empty);
  and  AND2_2(en_get, int2, req_get);
  and  AND2_3(int3, req_get, int1);
  and  AND2_4(valid_get, int2, int3);

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Token Controller
//
//------------------------------------------------------------------------

module tok_controller (tok_in, tok_control, tok_out);
  input  tok_in, tok_control;
  output tok_out;

  or OR2 (tok_out, tok_control, tok_in);

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Mixed-Clock FIFO Cell
//
//------------------------------------------------------------------------
//
module fifo_cell (rst, req_put, data_put, en_put, CLK_put, CLK_get, data_get, en_get,
           valid_i, ptok_out, ptok_prev, f_i, e_i, gtok_out, gtok_prev);

  parameter width = 2;

  input  [(width - 1):0] data_put;
  input  rst, req_put, en_put, CLK_put, CLK_get, en_get, ptok_prev, gtok_prev;
  output [(width - 1):0] data_get;
  output valid_i, ptok_out, gtok_out, f_i, e_i;

  wire  [(width - 1):0] data_out;
  wire  valid_bit, reg_en, get_en, gtok_ctrl, actual_gtok;

  enable_reg tokreg_in0  (rst, CLK_put, en_put, ptok_out, ptok_prev);
  enable_reg tokreg_in1  (rst, CLK_get, en_get, gtok_out, gtok_prev);

  data_register #(width) dreg_0(rst, CLK_put, data_put, req_put, reg_en,
                            get_en, data_out, valid_bit);

  and AND2_0  (reg_en, en_put, ptok_prev);
  and AND2_1  (get_en, en_get, gtok_prev);
  bufif1 tbuf_0 (valid_i , valid_bit , get_en);
  tri_state_bufn #(width) tbufn_0 (data_out , data_get , get_en);

  sr_ff srff0 (rst , reg_en , get_en , f_i , e_i);

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Generic N-bits Tri-state Buffer
//
//-----------------------------------------------------------------------

module tri_state_bufn (in_val , out_val , buf_en);

  parameter width = 2;

  input  [(width - 1):0] in_val;
  input                  buf_en;
  output [(width - 1):0] out_val;

  reg    [(width - 1):0] out_val;

  always @(buf_en or in_val)
    begin
    if (buf_en)
        out_val = in_val;
    else
        out_val = {(width){1'bz}};
    end

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// SR flip-flop
//
//-----------------------------------------------------------------------

module sr_ff (rst , set , reset , out_f , out_e);
  input  rst , set , reset;
  output out_f , out_e;

  reg out_f , out_e;

  always @(set or reset or rst or out_f or out_e)
    begin
    if (rst == 1)
      begin
        out_f = 0;
            out_e = 1;
      end
    else
      begin
        out_e = !(set | out_f);
        out_f = !(reset | out_e);
      end
    end
endmodule
```

```
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// Generic Latch
//
//---------------------------------------------------------------------------

module en_latch (nrst, en, out_val, in_val);

  parameter width = 1;
  parameter default_out = {(width){1'b0}};

  input   nrst, en;
  input   [(width - 1):0] in_val;
  output  [(width - 1):0] out_val;

  reg     [(width - 1):0] out_val;

  always @(in_val or en or nrst)
    begin
    if (nrst == 1)
      begin
                out_val = default_out;
      end
    else
      begin
        if (en == 1)
              begin
             out_val = in_val;
              end
        else
          begin
                 out_val = default_out;
          end
      end
    end

endmodule

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// Latch (retains pervious value when en = 0)
//
//---------------------------------------------------------------------------

module clk_latch (rst, en, out_val, in_val);

  parameter width = 1;
  parameter default_out = {(width){1'b0}};

  input   rst, en;
  input   [(width - 1):0] in_val;
  output  [(width - 1):0] out_val;

  reg     [(width - 1):0] out_val;
  reg     [(width - 1):0] temp_val;


  always @(in_val or en or rst or temp_val)
    begin
    if (rst == 1)
      begin
                out_val = default_out;
           temp_val = default_out;
      end
    else
      begin
        if (en == 1)
              begin
             temp_val = in_val;
              out_val = temp_val;
              end
        else
          begin
                  out_val = temp_val;
          end
      end
    end

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Edge Trigger Register
//
//------------------------------------------------------------------------

module register (rst, clk, out_val, in_val);

  parameter  in_val_width = 1;
  parameter  default_out  = {(in_val_width){1'b0}};

  input  clk, rst;
  input  [(in_val_width - 1):0] in_val;
  output [(in_val_width - 1):0] out_val;

  reg    [(in_val_width - 1):0] out_val;

  always @(posedge clk or posedge rst)
    begin
    if(rst == 1)
      begin
        out_val = default_out;
      end
    else
      begin
        out_val = in_val;
      end
    end

endmodule
```

```
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// Clock Edge Trigger Register with Enable
//
//---------------------------------------------------------------------------
//
module enable_reg (rst , clk , en, out_val , in_val);

  parameter width = 1;

  input  rst , clk , en;
  input  [(width - 1):0] in_val;
  output [(width - 1):0] out_val;

  reg    [(width - 1):0] temp_val;
  reg    [(width - 1):0] out_val;

  always @(posedge clk or posedge rst)
    begin
    if (rst == 1)
      begin
           temp_val = {(width){1'b0}};
                out_val = {(width){1'b0}};
      end
    else
      begin
        if (en == 1)
              begin
            temp_val = in_val;
             out_val = in_val;
              end
        else
          begin
                 out_val = temp_val;
          end
      end
    end

endmodule

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// Data Register of FIFO Cell
//
//---------------------------------------------------------------------------
//
module data_register (rst ,clk ,data ,req_put ,reg_en ,rst_flag ,out1 ,out2);

  parameter width = 2;

  input  [(width - 1):0] data;
  input  rst , clk , req_put , reg_en , rst_flag;
  output [(width - 1):0] out1;
  output out2;


  reg    [(width - 1):0] temp_data;
  reg                    temp_req_put;
  reg    [(width - 1):0] out1;
  reg                out2 ,rRst;


  always @(posedge clk or posedge rst)
    begin
    if ( rst == 1)
      begin
            out1 = 0;
            out2 = 0;
            temp_req_put = 0;
               temp_data = 0;
                 rRst = 0;
      end

    else
      begin
        if ( reg_en == 1)
               begin
                 temp_data = data;
                          out1 = temp_data;
              temp_req_put = req_put;
                          out2 = temp_req_put;
          end
          end
    end

endmodule
```

## A.2 Synthesizable Verilog Model of MCSB (using a 4-place MCFIFO)

```
//-----------------------------------------------------------------------------
//
//      This confidential and proprietary software may be used only
//    as authorized by the University of Alberta
//    In the event of publication , the following notice is applicable:
//
//                    (C) COPYRIGHT 2005  University of Alberta
//                           ALL RIGHTS RESERVED
//
//      The entire notice above must be reproduced on all authorized
//    copies.
//
// AUTHOR:      Edmund Fung                    LAST REVISION: May 26, 2005
//
// VERSION:     Simulation Architecture
//
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//
// Toplevel Module of SoC with MCSB
//
//-----------------------------------------------------------------------------
//
module toplevel_MCSB_SoC ( rst ,CLK_0,CLK_1,CLK_2,ptok_control ,gtok_control ,
                 int_addr ,go_0 ,go_1 ,go_2 ,en_get ,req_put_0 ,req_put_1 ,
                 req_put_2 ,en_put);

  parameter  bus_width       = 7;
  parameter  sel_width       = 2;
  parameter  target_addr_width  = 3;
  parameter  req_put_width   = 3;
  parameter  trigger_width   = 3;

  parameter  addr_width  = target_addr_width + req_put_width;

  input   rst ,CLK_0,CLK_1,CLK_2;
  input   go_0 ,go_1 ,go_2;
  input   ptok_control ,gtok_control;

  input   [(target_addr_width - 1):0] int_addr;

  output  en_get ,req_put_0 ,req_put_1 ,req_put_2 ,en_put;

  wire    req_put_0 ,req_put_1 ,req_put_2;
  wire    req_get_0 ,req_get_1 ,req_get_2;
  wire    valid_get_0 ,valid_get_1 ,valid_get_2;
  wire    full , empty;
  wire    [(bus_width - 1):0]    data_in_0;
  wire    [(bus_width - 1):0]    data_in_1;
  wire    [(bus_width - 1):0]    data_in_2;
  wire    [(bus_width - 1):0]    data_out_0;
  wire    [(bus_width - 1):0]    data_out_1;
  wire    [(bus_width - 1):0]    data_out_2;
  wire    [(target_addr_width - 1):0] target_addr_1 ,target_addr_2;
  wire    [(target_addr_width - 1):0] target_addr , target_addr_0;
  wire    [(trigger_width - 1):0]     trigger;
  wire    [(addr_width - 1):0]        address;

  assign address     = {go_2 ,go_1 ,go_0 ,target_addr};
  assign target_addr = (target_addr_0 | target_addr_1 | target_addr_2);
  SoC_module   #( bus_width , target_addr_width )
       mod_0    ( rst ,CLK_0, req_put_0 , req_get_0 , data_out_0 , data_in_0 ,
         valid_get_0 , trigger [0] , full ,empty , go_0 , int_addr ,
         target_addr_0 );
  SoC_module   #( bus_width , target_addr_width )
       mod_1    ( rst ,CLK_1 , req_put_1 , req_get_1 , data_out_1 , data_in_1 ,
         valid_get_1 , trigger [1] , full ,empty , go_1 , int_addr ,
         target_addr_1 );
  SoC_module   #( bus_width , target_addr_width )
       mod_2    ( rst ,CLK_2, req_put_2 , req_get_2 , data_out_2 , data_in_2 ,
         valid_get_2 , trigger [2] , full ,empty , go_2 , int_addr ,
         target_addr_2 );

  toplevel_MCSB #( bus_width , sel_width , target_addr_width ,
          req_put_width , trigger_width )
     MCSB_0  ( rst , req_put_0 , req_put_1 , req_put_2 , req_get_0 , req_get_1 ,
          req_get_2 ,CLK_0,CLK_1,CLK_2, ptok_control , gtok_control ,
          target_addr , data_in_0 , data_in_1 , data_in_2 , data_out_0 ,
          data_out_1 , data_out_2 , full ,empty , valid_get_0 , valid_get_1 ,
          valid_get_2 , en_get , trigger , address , en_put );

endmodule

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//
// Toplevel Module of MCSB
//
//-----------------------------------------------------------------------------
//
module toplevel_MCSB ( rst , req_put_0 , req_put_1 , req_put_2 , req_get_0 , req_get_1 ,
              req_get_2 ,CLK_0,CLK_1,CLK_2, ptok_control , gtok_control ,
              target_addr , data_in_0 , data_in_1 , data_in_2 , data_out_0 ,
              data_out_1 , data_out_2 , full ,empty , valid_get_0 , valid_get_1 ,
              valid_get_2 , en_get , trigger , address , en_put );

  parameter  bus_width       = 6;
  parameter  sel_width       = 2;
  parameter  target_addr_width   = 3;
  parameter  req_put_width   = 3;
  parameter  trigger_width   = 3;

  parameter  addr_width   = target_addr_width + req_put_width;

  input  rst , req_put_0 , req_put_1 , req_put_2 , req_get_0 , req_get_1 , req_get_2;
  input  CLK_0,CLK_1,CLK_2, ptok_control , gtok_control;

  input   [( addr_width - 1):0]     address;
  input   [( req_put_width  - 1):0] target_addr;
  input   [( bus_width - 1):0]    data_in_0;
  input   [( bus_width - 1):0]    data_in_1;
  input   [( bus_width - 1):0]    data_in_2;

  output  valid_get_0 , valid_get_1 , valid_get_2 , full ,empty , en_get , en_put;

  output  [( trigger_width  - 1):0] trigger;
  output  [( bus_width - 1):0]    data_out_0;
  output  [( bus_width - 1):0]    data_out_1;
  output  [( bus_width - 1):0]    data_out_2;

  wire    req_put , req_get , valid_get;
  wire    CLK_put, CLK_get;

  wire    [( sel_width  - 1):0] fmsel , rmsel , rdsel;
  wire    [( addr_width  - 1):0] address;
  wire    [( trigger_width  - 1):0] trigger;
  wire   [(bus_width - 1):0] data_put;
  wire   [(bus_width - 1):0] data_get;

  mux3to1 #(bus_width)
     m0(data_in_0, data_in_1, data_in_2, fmsel, data_put);
  mux3to1 m1(req_put_0, req_put_1, req_put_2, fmsel, req_put);
  mux3to1 m2(CLK_0,CLK_1,CLK_2, fmsel, CLK_put);

  mux3to1 m3(req_get_0, req_get_1, req_get_2, rmsel, req_get);
  mux3to1 m4(CLK_0,CLK_1,CLK_2, rmsel, CLK_get);

  demux1to3 #(bus_width)
       de0(rst, data_get, rdsel, data_out_0, data_out_1, data_out_2);
  demux1to3 de1(rst, valid_get, rdsel, valid_get_0, valid_get_1, valid_get_2);


  toplevel_MCFIFO  #(bus_width)
            MCFIFO_0    (rst, full, empty, req_put, data_put, CLK_put,
                   ptok_control, CLK_get, data_get, req_get,
                       valid_get, gtok_control, en_get, en_put);

  MCSB_bus_controller    bctrl0 (rst, address, trigger, fmsel, rmsel, rdsel);

endmodule

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//
// 3-to-1 Multiplexer
//
//---------------------------------------------------------------------------
//
module mux3to1 (A,B,C, sel,Z);

  parameter width = 1;
  parameter sel_width = 2;

  input  [(width - 1):0] A,B,C;
  input  [(sel_width - 1):0] sel;
  output [(width - 1):0] Z;

  reg    [(width - 1):0] Z;

  always @(sel or A or B or C)
    begin
    if (sel == 2'b00)
      begin
        Z = A;
      end
    else
      begin
        if (sel == 2'b10)
          begin
            Z = B;
          end
        else
          begin
            if (sel == 2'b01)
              begin
                Z = C;
              end
          end
      end
    end

endmodule
```

```
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//
// 1 - to -3 Decoder
//
//----------------------------------------------------------------------------
//
module dec1to3 ( rst ,A, sel ,Out1 ,Out2 ,Out3 );

  parameter width = 1;
  parameter sel_width = 2;

  input   rst;
  input   [(width - 1):0] A;
  input   [(sel_width - 1):0] sel;
  output  [(width - 1):0] Out1 , Out2 , Out3;

  reg     [(width - 1):0] Out1 , Out2 , Out3;

always @(rst)
  begin
    if (rst == 1)
      begin
        Out1 = 'bz;
        Out2 = 'bz;
        Out3 = 'bz;
      end
  end


always @(A or sel)
  begin
        if (sel == 2'b00)
      begin
        Out1 = A;
        Out2 = 'bz;
        Out2 = 'bz;
      end
    else
      begin
        if (sel == 2'b10)
          begin
            Out1 = 'bz;
            Out2 = A;
            Out3 = 'bz;
          end
        else
          begin
            if (sel == 2'b01)
              begin
                Out1 = 'bz;
                Out2 = 'bz;
                Out3 = A;
              end
          end
      end
  end

endmodule
```

```
//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Bus Controller of MCSB
//
//------------------------------------------------------------------------
//
module bus_controller (rst, address, trigger, fmsel, rmsel, rdsel);

  parameter addr_width      = 6;
  parameter trigger_width   = 3;
  parameter sel_width       = 2;
  parameter out_width       = (3*sel_width);

  input  rst;
  input  [(addr_width - 1):0]    address;

  output [(trigger_width - 1):0] trigger;
  output [(sel_width - 1):0]     fmsel, rmsel, rdsel;

  wire   [(sel_width - 1):0]     fmsel, rmsel, rdsel;
  wire   [(trigger_width - 1):0] trigger;
  reg    [(out_width - 1):0]     controller_out;

  assign trigger = {address[2], address[1], address[0]};
  assign fmsel   = {controller_out[5], controller_out[4]};
  assign rmsel   = {controller_out[3], controller_out[2]};
  assign rdsel   = {controller_out[1], controller_out[0]};

  always @(posedge rst)
    begin
    controller_out = 'b0;
    end

  always @(address)
    begin
    case(address)
      6'b100010 : controller_out = 6'b011010;   // Module 0
      6'b100001 : controller_out = 6'b010000;
      6'b010100 : controller_out = 6'b100101;   // Module 1
      6'b010001 : controller_out = 6'b100000;
      6'b001100 : controller_out = 6'b000101;   // Module 2
      6'b001010 : controller_out = 6'b001010;
        default : controller_out = 6'b000000;
    endcase
    end

endmodule

//------------------------------------------------------------------------
//------------------------------------------------------------------------
//
// Toplevel Module of the Mixed-Clock FIFO
//
//------------------------------------------------------------------------
//
module toplevel_MCFIFO (rst, full, empty, req_put, data_put, CLK_put, ptok_control,
                        CLK_get, data_get, req_get, valid_get, gtok_control, en_get);

  parameter width = 7;

  input                       rst, req_put, CLK_put, CLK_get, req_get;
  input                       ptok_control, gtok_control;
  input  [(width - 1):0] data_put;
  output [(width - 1):0] data_get;
  output                      full, empty, valid_get, en_get;
  wire                        en_put , en_get;
  wire                        valid;
  wire                        valid_i;

  wire      [3:0]             cell_full , cell_empty;
  wire                        ne , oe;
  wire      [3:0]             ptok_in , gtok_in , ptok_out , gtok_out;

  assign   ptok_in[1] = ptok_out[0];
  assign   ptok_in[2] = ptok_out[1];
  assign   ptok_in[3] = ptok_out[2];

  assign   gtok_in[1] = gtok_out[0];
  assign   gtok_in[2] = gtok_out[1];
  assign   gtok_in[3] = gtok_out[2];


  full_detector    fd_0 (rst ,cell_empty[0],cell_empty[1],cell_empty[2],
                         cell_empty[3],CLK_put, full);

  empty_detector  ed_0 (rst ,cell_full[0],cell_full[1],cell_full[2],
                         cell_full[3],CLK_get,en_get,oe,ne);


  put_controller  pc_0 (full ,req_put ,en_put);

  get_controller  gc_0 (req_get ,ne,oe, valid_i ,en_get , valid ,empty);

  en_latch         valid_latch_0 (rst , valid_i , valid_get , valid);


  fifo_cell       #(width)
          ff_0(rst ,req_put ,data_put ,en_put ,CLK_put,CLK_get,data_get ,
                        en_get , valid_i ,ptok_out[0],ptok_in[0],cell_full[0],
                        cell_empty[0],gtok_out[0],gtok_in[0]);
  fifo_cell       #(width)
          ff_1(rst ,req_put ,data_put ,en_put ,CLK_put,CLK_get,data_get ,
                        en_get , valid_i ,ptok_out[1],ptok_in[1],cell_full[1],
                        cell_empty[1], gtok_out[1], gtok_in[1]);
  fifo_cell       #(width)
          ff_2(rst ,req_put ,data_put ,en_put ,CLK_put,CLK_get,data_get ,
                        en_get , valid_i ,ptok_out[2],ptok_in[2],cell_full[2],
                        cell_empty[2],gtok_out[2],gtok_in[2]);
  fifo_cell       #(width)
          ff_3(rst ,req_put ,data_put ,en_put ,CLK_put,CLK_get,data_get ,
                        en_get , valid_i ,ptok_out[3],ptok_in[3],cell_full[3],
                        cell_empty[3],gtok_out[3],gtok_in[3]);


  tok_controller  ptc_0(ptok_out[3],ptok_control ,ptok_in[0]);
  tok_controller  gtc_0(gtok_out[3],gtok_control ,gtok_in[0]);

endmodule
```

```
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//
// Full Detector
//
//----------------------------------------------------------------------------

module full_detector (rst, e_0, e_1, e_2, e_3, CLK_put, full);
  input  rst, e_0, e_1, e_2, e_3, CLK_put;
  output full;

  wire   reg_buf, not_1_out, not_0_out;
  wire   not_e_0, not_e_1, not_e_2, not_e_3;
  wire   A, B, C, D, E, F, pre_out;

  not not_0  (not_e_0,e_0);
  not not_1  (not_e_1,e_1);
  not not_2  (not_e_2,e_2);
  not not_3  (not_e_3,e_3);

  or  or2_0  (A,not_e_0,not_e_1);
  or  or2_1  (B,not_e_1,not_e_2);
  or  or2_2  (C,not_e_2,not_e_3);
  or  or2_3  (D,not_e_3,not_e_0);

  and and2_0 (E,A,B);
  and and2_1 (F,C,D);
  and and2_2 (pre_out,E,F);

  not not_4 (not_0_out, pre_out);
  not not_5 (not_1_out, not_0_out);
  register reg_0 (rst,CLK_put,reg_buf,not_1_out);
  register reg_1 (rst,CLK_put,full,reg_buf);

endmodule

//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//
// Toplevel Module of Empty Detector
//
//----------------------------------------------------------------------------

module empty_detector (rst, f_0, f_1, f_2, f_3, CLK_get, en_get, oe, ne);
  input  rst, f_0, f_1, f_2, f_3, CLK_get, en_get;
  output oe, ne;

  normal_empty_detector NED_0 (rst, f_0, f_1, f_2, f_3, CLK_get, ne);
  true_empty_detector   TED_0 (rst, f_0, f_1, f_2, f_3, CLK_get, en_get, oe);

endmodule

//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//
// Normal Empty Detector
//
//----------------------------------------------------------------------------

module normal_empty_detector (rst, f_0, f_1, f_2, f_3, CLK_get, ne);
  input  rst, f_0, f_1, f_2, f_3, CLK_get;
  output ne;

  parameter  in_width = 1;
  parameter  default_out  = {(in_width){1'b1}};
  wire reg_buf , not_0_out , not_1_out , pre_out;
  wire A, B, C, D, E, F, G, H;

  and and2_0 (A, f_0 , f_1 );
  and and2_1 (B, f_1 , f_2 );
  and and2_2 (C, f_2 , f_3 );
  and and2_3 (D, f_3 , f_0 );

  or  or2_0  (E,A,B);
  or  or2_1  (F,C,D);
  or  or2_2  (G,E,F);

  not not_0  (pre_out ,G);
  not not_1 ( not_0_out , pre_out);
  not not_2 ( not_1_out , not_0_out );
  register  #(in_width , default_out)
        reg_0 ( rst ,CLK_get, reg_buf , not_1_out );
  register  #(in_width , default_out)
        reg_1 ( rst ,CLK_get, ne , reg_buf );

endmodule

//-----------------------------------------------------------------
//-----------------------------------------------------------------
//
// True Empty Detector
//
//-----------------------------------------------------------------

module true_empty_detector (rst , f_0 , f_1 , f_2 , f_3 , CLK_get , en_get , oe);
  input  rst , f_0 , f_1 , f_2 , f_3 , CLK_get , en_get;
  output oe;

  parameter  in_width = 1;
  parameter  default_out   = {(in_width){1'b1}};

  wire   pre_out , A, B, C;
  wire   reg_buf , prev_oe;

         or OR2_A (A, f_0 , f_1 );
      or OR2_B (B, f_2 , f_3 );
      or OR2_C (C,A,B);
     not not_1 ( pre_out ,C);

    register #(in_width , default_out)
         reg_0 ( rst ,CLK_get, reg_buf , pre_out );
          or OR2_0 ( prev_oe , reg_buf , en_get );
    register #(in_width , default_out)
         reg_1 ( rst ,CLK_get, oe , prev_oe );

endmodule

//-----------------------------------------------------------------
//-----------------------------------------------------------------
//
// Put Controller
//
//-----------------------------------------------------------------

module put_controller ( full , req_put , en_put);
  input  full , req_put;
  output en_put;

  reg nfull , en_put;

  always @( req_put or full)
```

```
    begin
    nfull = !( full );
    en_put = req_put & nfull ;
    end

endmodule

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//
// Get Controller
//
//--------------------------------------------------------------------------------

module get_controller (req_get , ne, oe, valid_i , en_get , valid_get , empty);
  input   req_get , ne, oe, valid_i ;
  output  en_get , valid_get , empty;

  wire    int1 , int2 , int3 ;
  assign  int1 = ~valid_i ;

  and  AND2_0(empty , ne , oe );
  not  NOT_0( int2 , empty );
  and  AND2_2( en_get , int2 , req_get );
  and  AND2_3( int3 , req_get , int1 );
  and  AND2_4( valid_get , int2 , int3 );

endmodule

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//
// Token Controller
//
//--------------------------------------------------------------------------------

module tok_controller (tok_in , tok_control , tok_out);
  input  tok_in , tok_control ;
  output tok_out ;

  or OR2 ( tok_out , tok_control , tok_in );

endmodule

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//
// Mixed-Clock FIFO Cell
//
//--------------------------------------------------------------------------------
//
module fifo_cell (rst ,req_put ,data_put ,en_put ,CLK_put ,CLK_get ,data_get ,en_get ,
          valid_i ,ptok_out ,ptok_prev ,f_i ,e_i ,gtok_out ,gtok_prev );

  parameter width = 2;

  input  [(width - 1):0] data_put ;
  input  rst ,req_put ,en_put ,CLK_put ,CLK_get ,en_get ,ptok_prev ,gtok_prev ;
  output [(width - 1):0] data_get ;
  output valid_i ,ptok_out ,gtok_out ,f_i ,e_i ;

  wire  [(width - 1):0] data_out ;
  wire  valid_bit ,reg_en ,get_en ,gtok_ctrl ,actual_gtok ;

  enable_reg tokreg_in0  (rst , CLK_put , en_put , ptok_out , ptok_prev );
  enable_reg tokreg_in1  (rst , CLK_get , en_get , gtok_out , gtok_prev );
  data_register #(width) dreg_0(rst, CLK_put, data_put, req_put, reg_en,
                                get_en, data_out, valid_bit);

  and AND2_0  (reg_en, en_put, ptok_prev);
  and AND2_1  (get_en, en_get, gtok_prev);

  bufif1 tbuf_0 (valid_i, valid_bit, get_en);
  tri_state_bufn #(width) tbufn_0 (data_out, data_get, get_en);

  sr_ff srff0 (rst, reg_en, get_en, f_i, e_i);

endmodule

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// Generic N-bits Tri-state Buffer
//
//--------------------------------------------------------------------------

module tri_state_bufn (in_val, out_val, buf_en);

  parameter width = 2;

  input  [(width - 1):0] in_val;
  input                  buf_en;
  output [(width - 1):0] out_val;

  reg    [(width - 1):0] out_val;

  always @(buf_en or in_val)
    begin
    if (buf_en)
        out_val = in_val;
    else
        out_val = {(width){1'bz}};
    end

endmodule

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
//
// SR flip-flop
//
//--------------------------------------------------------------------------

module sr_ff (rst, set, reset, out_f, out_e);
  input  rst, set, reset;
  output out_f, out_e;

  reg out_f, out_e;

  always @(set or reset or rst or out_f or out_e)
    begin
    if (rst == 1)
      begin
        out_f = 0;
            out_e = 1;
      end
    else
      begin
        out_e = !(set | out_f);
        out_f = !(reset | out_e);
      end
    end
endmodule

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//
// Generic Latch
//
//-----------------------------------------------------------------------------

module en_latch (nrst, en, out_val, in_val);

  parameter width = 1;
  parameter default_out = {(width){1'b0}};

  input  nrst, en;
  input  [(width - 1):0] in_val;
  output [(width - 1):0] out_val;

  reg    [(width - 1):0] out_val;

  always @(in_val or en or nrst)
    begin
    if (nrst == 1)
      begin
                out_val = default_out;
      end
    else
      begin
        if (en == 1)
              begin
             out_val = in_val;
              end
        else
          begin
                  out_val = default_out;
          end
      end
    end

endmodule

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//
// Edge Trigger Register
//
//-----------------------------------------------------------------------------

module register (rst, clk, out_val, in_val);

  parameter  in_val_width = 1;
  parameter  default_out  = {(in_val_width){1'b0}};

  input  clk, rst;
  input  [(in_val_width - 1):0] in_val;
  output [(in_val_width - 1):0] out_val;

  reg    [(in_val_width - 1):0] out_val;

  always @(posedge clk or posedge rst)
    begin
    if(rst == 1)
      begin
        out_val = default_out;
      end
    else
      begin
        out_val = in_val;
      end
    end

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Clock Edge Trigger Register with Enable
//
//-----------------------------------------------------------------------
//
module enable_reg (rst , clk , en , out_val , in_val);

  parameter width = 1;

  input  rst , clk , en;
  input  [(width - 1):0] in_val;
  output [(width - 1):0] out_val;

  reg    [(width - 1):0] temp_val;
  reg    [(width - 1):0] out_val;

  always @(posedge clk or posedge rst)
    begin
    if (rst == 1)
      begin
           temp_val = {(width){1'b0}};
                out_val = {(width){1'b0}};
      end
    else
      begin
        if (en == 1)
              begin
            temp_val = in_val;
             out_val = in_val;
              end
        else
          begin
                out_val = temp_val;
          end
      end
    end

endmodule

//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
//
// Data Register of FIFO Cell
//
//-----------------------------------------------------------------------
//
module data_register (rst ,clk ,data ,req_put ,reg_en ,rst_flag ,out1 ,out2);

  parameter width = 2;

  input  [(width - 1):0] data;
  input  rst , clk , req_put , reg_en , rst_flag;
  output [(width - 1):0] out1;
  output out2;
```

```
reg     [(width  - 1):0] temp_data;
reg                      temp_req_put;
reg     [(width  - 1):0] out1;
reg                  out2 , rRst;


always @(posedge clk or posedge rst)
  begin
  if ( rst == 1)
    begin
          out1  = 0;
          out2  = 0;
          temp_req_put  = 0;
             temp_data  = 0;
              rRst  = 0;
    end

  else
    begin
      if ( reg_en == 1)
            begin
              temp_data  = data;
                       out1  = temp_data;
           temp_req_put  = req_put;
                       out2  = temp_req_put;
        end
        end
  end

endmodule
```

# Appendix B

# Simulation Waveforms

## B.1 Simulations of DBSB

Figure B.1 illustrates the SoC model and its configuration used to verify the functionality of the DBSB bus. The clock ratios are selected such that a data item sends by Module 0 is broken down into four segments by a type-B 1-to-4 shift register but reunites into a single data item through a type-A 4-to-1 shift register. Thus the data item identical to the one being sent should appear at *data_out_2* of Module 2. The same concept applies to the timing relationship between Module 1 and Module 3. The function of each test case for the DBSB is described in Chapter 4. Module 0 operates at 20.83MHz and Module operates at 41.67MHz. Module 2 operates at 25.00MHz while Module operates at 50.00MHz.*CLK_put* of MCFIFO_0 is 83.33MHz and *CLK_get* of MCFIFO_0 is 100MHz. On the other hand, *CLK_put* of MCFIFO_1 is 100.00MHz and *CLK_get* of MCFIFO_1 is at 83.33MHz.

## B.2 Simulations of MCSB

Figure 4.7 illustrates the SoC model and its configuration used to verify the functionality of the MCSB bus. The function of each test case for the MCSB is described in Chapter 4. Module 0 operates at 83.33MHz. Module 1 operates at 41.67MHz. Module 2 operates at 20.83MHz.

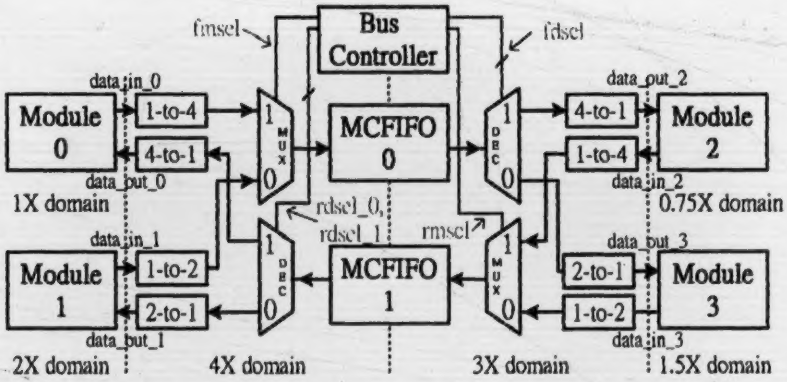

fmsel
Bus Controller
fdsel
data_in_0
Module 0
1-to-4
4-to-1
MUX
1
0
MCFIFO 0
DEC
1
0
4-to-1
1-to-4
data_out_2
Module 2
data_out_0
1X domain
rdsel_0, rdsel_1
rmsel
data_in_2
0.75X domain
data_in_1
Module 1
1-to-2
2-to-1
DEC
1
0
MCFIFO 1
MUX
1
0
2-to-1
1-to-2
data_out_3
Module 3
data_out_1
data_in_3
2X domain
4X domain
3X domain
1.5X domain

Figure B.1: System Configuration Used in DBSB Simulations

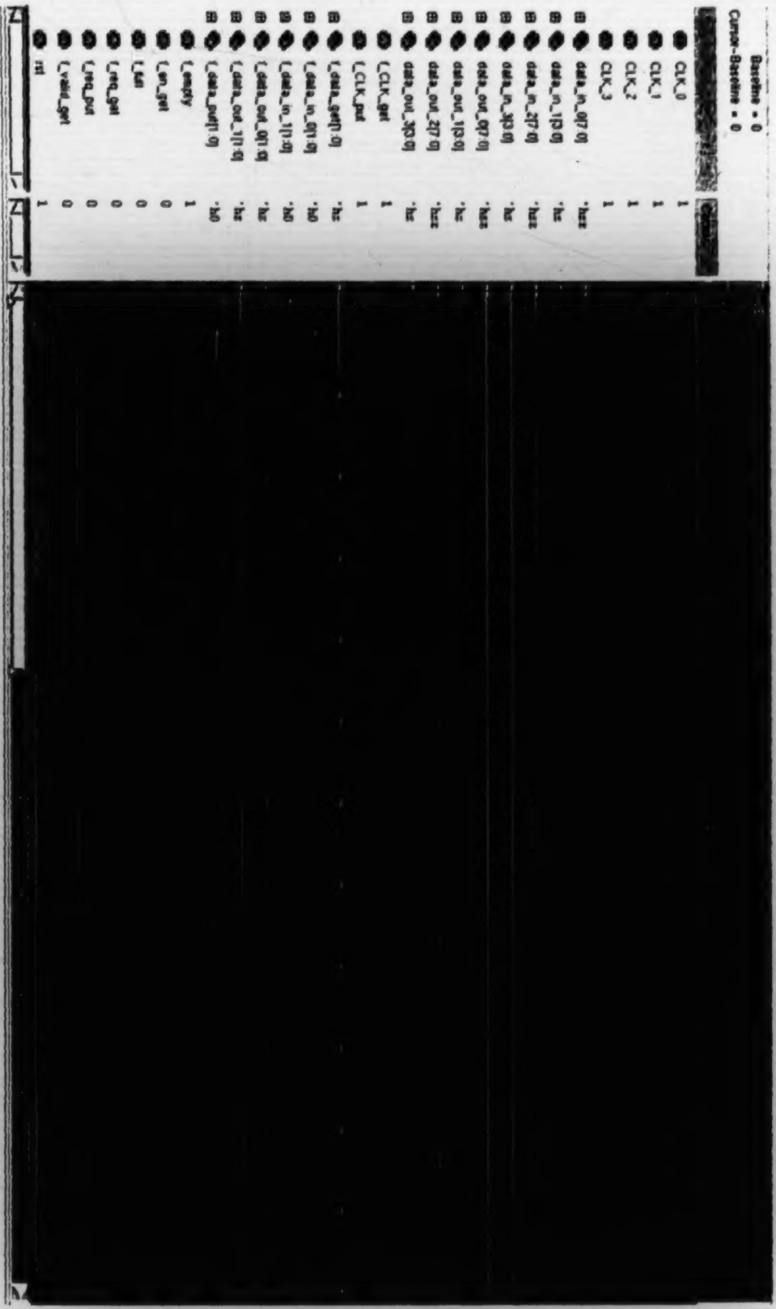

Baseline = 0
Cursor-Baseline = 0
CLK_0
CLK_1
CLK_2
CLK_3
data_in_0[7:0]
data_in_1[3:0]
data_in_2[7:0]
data_in_3[3:0]
data_out_0[7:0]
data_out_1[3:0]
data_out_2[7:0]
data_out_3[3:0]
I_CLK_get
I_CLK_put
I_data_get[1:0]
I_data_in_0[1:0]
I_data_in_1[1:0]
I_data_out_0[1:0]
I_data_out_1[1:0]
I_data_put[1:0]
I_empty
I_en_get
I_full
I_req_get
I_req_put
I_valid_get
rst
1
1
1
1
'hzz
'hz
'hzz
'hz
'hzz
'hz
'hzz
'hz
1
1
'hz
'h0
'h0
'h0
'hz
'h0
1
0
0
0
0
0
1

Figure B.2: Waveform of Test Case 1 of DSBS

Baseline = 0
Cursor-Baseline = 0
CLK_0 1
CLK_1 1
CLK_2 1
CLK_3 1
data_in_0[7:0] 'hzz
data_in_1[3:0] 'hz
data_in_2[7:0] 'hzz
data_in_3[3:0] 'hz
data_out_0[7:0] 'hzz
data_out_1[3:0] 'hz
data_out_2[7:0] 'hzz
data_out_3[3:0] 'hz
f_CLK_get 1
f_CLK_put 1
f_data_get[1:0] 'hz
f_data_in_0[1:0] 'h0
f_data_in_1[1:0] 'h0
f_data_out_0[1:0] 'hz
f_data_out_1[1:0] 'hz
f_data_put[1:0] 'h0
f_empty 1
f_en_get 0
f_full 0
f_req_get 0
f_req_put 0
f_valid_get 0
rst 1

Figure B.3: Waveform of Test Case 2 of DSBS

Baseline = 0
Cursor-Baseline = 0
CLK_0 1
CLK_1 1
CLK_2 1
CLK_3 1
data_in_0[7:0] 'hzz
data_in_1[3:0] 'hz
data_in_2[7:0] 'hzz
data_in_3[3:0] 'hz
data_out_0[7:0] 'hzz
data_out_1[3:0] 'hz
data_out_2[7:0] 'hzz
data_out_3[3:0] 'hz
f_CLK_get 1
f_CLK_put 1
f_data_get[1:0] 'hz
f_data_in_0[1:0] 'b0
f_data_in_1[1:0] 'b0
f_data_out_0[1:0] 'hz
f_data_out_1[1:0] 'hz
f_data_put[1:0] 'b0
f_empty 1
f_en_get 0
f_full 0
f_req_get 0
f_req_put 0
f_valid_get 0
rst 1
Zone 1
Zone 2

Figure B.4: Waveform of Test Case 3 of DSBS

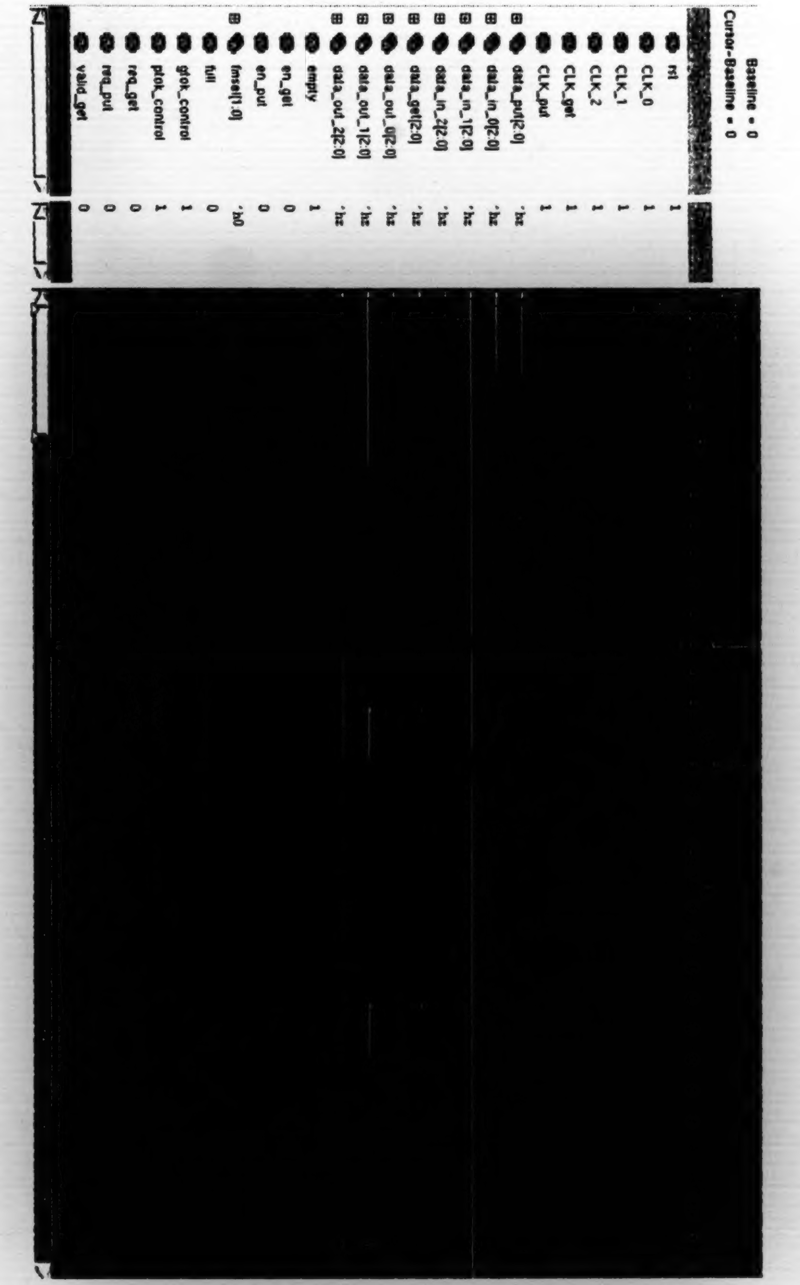

Baseline = 0
Cursor-Baseline = 0
rst 1
CLK_0 1
CLK_1 1
CLK_2 1
CLK_get 1
CLK_put 1
data_put[2:0] 'hz
data_in_0[2:0] 'hz
data_in_1[2:0] 'hz
data_in_2[2:0] 'hz
data_get[2:0] 'hz
data_out_0[2:0] 'hz
data_out_1[2:0] 'hz
data_out_2[2:0] 'hz
empty 1
en_get 0
en_put 0
fmsel[1:0] 'h0
full 0
glob_control 1
ptok_control 1
req_get 0
req_put 0
valid_get 0

Figure B.5: Waverform of Test Case 1 of MCSB

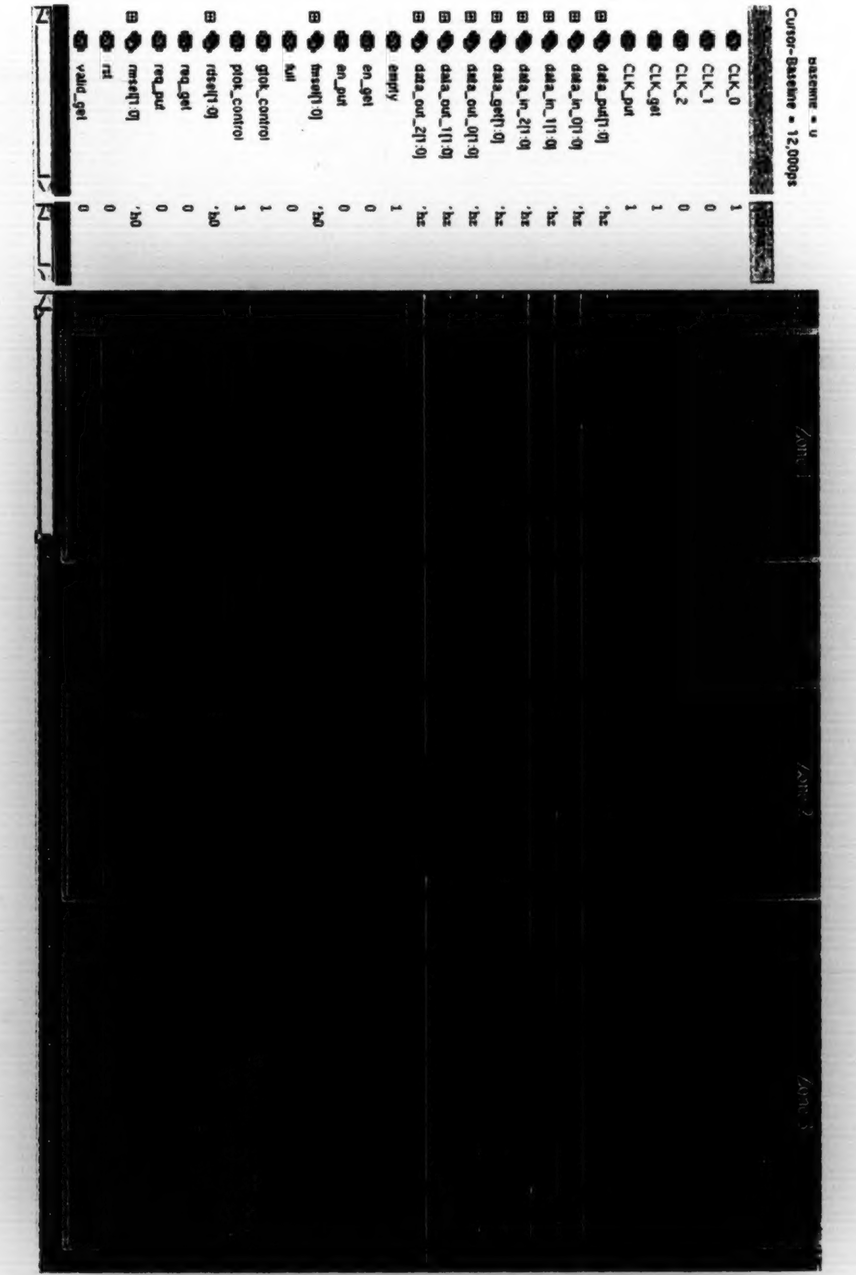

Baseline = 0
Cursor-Baseline = 12,000ps
CLK_0 1
CLK_1 0
CLK_2 0
CLK_get 1
CLK_put 1
data_put[1:0] 'hz
data_in_0[1:0] 'hz
data_in_1[1:0] 'hz
data_in_2[1:0] 'hz
data_get[1:0] 'hz
data_out_0[1:0] 'hz
data_out_1[1:0] 'hz
data_out_2[1:0] 'hz
empty 1
en_get 0
en_put 0
fmsel[1:0] 'h0
full 0
glob_control 1
plot_control 1
rdsel[1:0] 'h0
req_get 0
req_put 0
rmsel[1:0] 'h0
rst 0
valid_get 0
Zone 1
Zone 2
Zone 3

Figure B.6: Waveform of Test Case 2 of MCSB